pump框架设计文档

作者：杨铮

目录

[1 pump\_Event库 4](#_Toc53515848)

[1.1 业务、请求/响应、服务、事务及事件 4](#_Toc53515849)

[1.1.1 请求与响应 4](#_Toc53515850)

[1.1.2 服务 6](#_Toc53515851)

[1.1.3 事务 7](#_Toc53515852)

[1.1.4 事件 8](#_Toc53515853)

[1.2 服务模型 9](#_Toc53515854)

[1.2.1 Daemon服务 11](#_Toc53515855)

[1.3 事务模型 12](#_Toc53515856)

[1.3.1 需求分析 12](#_Toc53515857)

[1.3.2 详细设计 14](#_Toc53515858)

[1.4 协议架构 15](#_Toc53515859)

[1.5 事件驱动模型 15](#_Toc53515860)

[1.5.1 需求分析1.0（预研版） 16](#_Toc53515861)

[1.5.2 需求分析2.0（决定版） 26](#_Toc53515862)

[1.5.3 概要设计 29](#_Toc53515863)

[1.5.4 详细设计 35](#_Toc53515864)

提纲

1. pump\_core
2. pump\_memory
3. pump\_function
4. pump\_Event
5. pump\_core库
   1. 跨平台设计
   2. 全局资源管理模型
      1. 问题

在c++程序启动过程中，往往需要实例化很多全局单例对象。由于全局实例的初始化顺序由编译器决定，这表示默认情况下，全局实例的初始化顺序不确定的。一旦单例数量变多，并且存在单例之间的依赖的情况下，很容易造成A依赖B，但使用A时B不一定已经初始化。这个问题很容易造成程序偶发性的启动失败，甚至运行异常。

* + 1. 解决方案

引入全局资源管理模式，将所有原来的全局变量单例成为CGlobalCtrlBase的成员变量，然后由CGlobalCtrlBase统一初始化，即可让单例资源的初始化顺序受控。当然CGlobalCtrlBase是一个全局变量单例。

不同的独立模块一般需要初始化特定的单例资源，因此框架提供CGlobalCtrlBase类作为全局资源管理的基类。提供全局资源管理公共方法。

* + 1. 设计

在设计CGlobalCtrlBase时，因为是单例并且为了节省内存和方便模块的卸载，为单例分配堆上内存。因此需要考虑多线程情况下，会不产生竞态条件。

使用堆上单例，首先需要解决，多线程重复初始化/重复释放的问题，可能导致内存泄漏/非法内存释放；并且当有线程正在引用单例指针时，不允许其他线程释放单例堆内存，同时还要允许多线程并发引用单例指针。因此当引用单例堆内存时，需要上读锁，允许多个线程并发访问；而当给单例分配堆内存或者释放单例堆内存时，需要同时上写锁。

因为日志是模块的公共功能，所以在基类中包含::Pump::Core::CLogRecorderBase成员。

另外一个问题，当CGlobalCtrlBase的成员指针正被某个线程引用的时候，不允许其他线程销毁释放CGlobalCtrlBase，并且也不允许其他线程释放该成员指针。

因此引入resource keeper设计模式，所以CGlobalCtrlBase及其派生类中的单例指针资源在被线程引用时不允许直接返回裸指针，必须返回CGlobalResouceKeeper对象或其派生对象（注：CGlobalResouceKeeper允许拷贝）。

以CGlobalCtrlBase全局日志资源为例m\_pLogRecorder。日志资源时最常被外部引用的资源，按照resource keeper设计模式，从CGlobalResouceKeeper<CLogRecorderBase>派生，并实现构造和析构函数，在构造函数中为资源上读锁，关闭资源的销毁权限。

CGlobalCtrlBase成员：

|  |
| --- |
| pump\_bool\_t **m\_bInit**; |
| ::Pump::Core::Thread::CMutex **m\_csInit**; |
| ::Pump::Core::CLogRecorderBase \* **m\_pLogRecorder**; ///< private log recorder, [Not Owner] |
| ::Pump::Core::Thread:: CRWLocker **m\_csLogRecorder**; ///< log recorder locker，所有单例指针资源都必须配一个专属的读写锁，访问上读锁，修改上写锁 |
| static CGlobalCtrlBase \* **s\_pGlobalCtrl**; |
| /\*\*  \* Global instance RW locker. <b>s\_rcsGlobalCtrl<\b> prevent to free s\_pGlobalCtrl \  \* while other threads reference s\_pGlobalCtrl.  \* - When using s\_pGlobalCtrl on read locker.  \* - When assign/delete s\_pGlobalCtrl on write locker.  \*/  static ::Pump::Core::Thread::CRWLocker **s\_rcsGlobalCtrl**; |

CGlobalCtrlBase方法：

|  |
| --- |
| void **SetInitFlag**(pump\_bool\_t bInit); |
| static pump\_bool\_t **IsInit**(); |
| static pump\_int32\_t **SetLogger**(pump\_handle\_t hLogger); |
| /\*\*  \* 获取  \*/  static CLogRecorderGuider **GetLogger**(); |
| static void **ReadLock**(); |
| static void **ReadUnlock**(); |
| static void **WriteLock**(); |
| static void **WriteUnlock**(); |

CGlobalResouceKeeper成员：

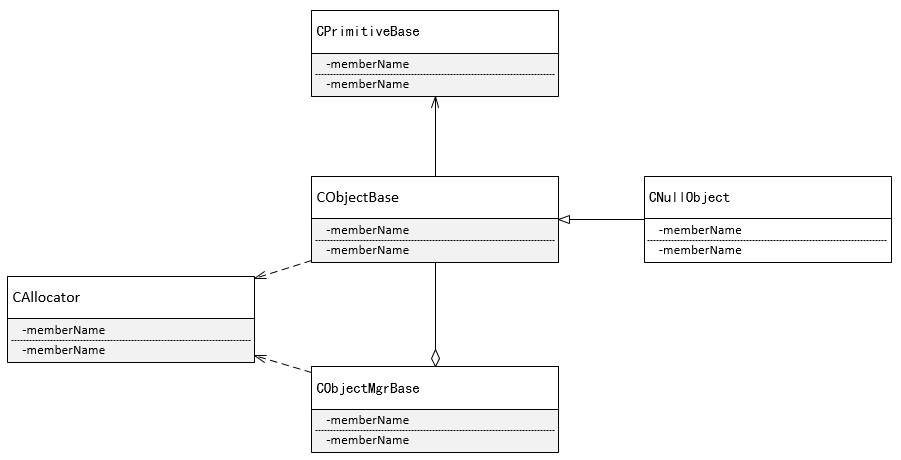
|  |
| --- |
| template <class T> |
| T \* **m\_ptr**; ///< 缓存s\_pGlobalCtrl中单例资源指针 |

CGlobalResouceKeeper成员：

|  |
| --- |
| /\*\*调用CGlobalCtrlBase::ReadLock() ，关闭赋值/释放全局资源管理单例s\_pGlobalCtrl的权限 \*/  explicit **CGlobalResouceKeeper**(T \* ptr) |
| /\*\*调用CGlobalCtrlBase::ReadUnlock() ，开启赋值/释放全局资源管理单例s\_pGlobalCtrl的权限 \*/  virtual ~**CGlobalResouceKeeper**() |
| /\*\*调用CGlobalCtrlBase::ReadLock() ，关闭赋值/释放全局资源管理单例s\_pGlobalCtrl的权限 \*/  **CGlobalResouceKeeper**(const CGlobalResouceKeeper & other) |
| T \* **GetPtr**() |

* 1. 对象管理模型

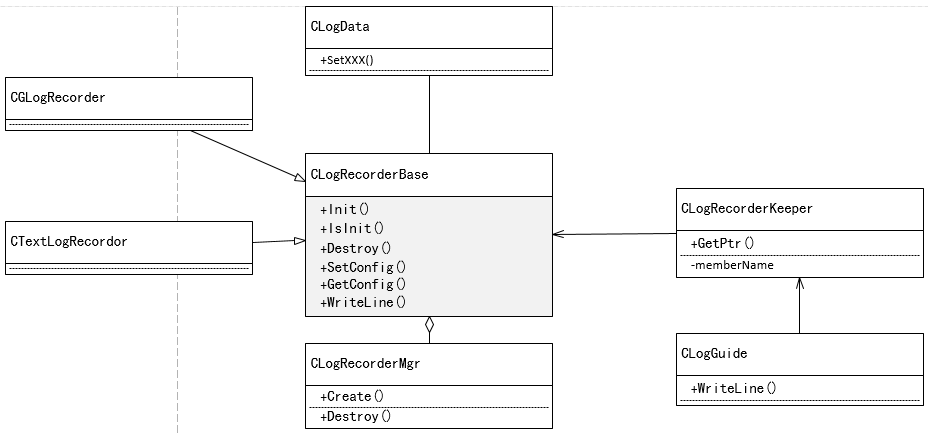
对象全局管理模型设计的目标是，统一管理应用程序内部对象的栈上内存，统一分配、并在程序退出时统一释放，避免内存泄露和内存碎片。属于基础框架层模型，使用抽象工厂模式设计。



* 1. io 模型
     1. 日志框架
        1. 问题

开发框架为了适配多种日志后端，需要将日志的公共流程梳理出来。比如开发者可以选择使用boost的日志库或者glog、log4，只需要编译框架时启动相应编译选项。甚至，开发者在日志框架的基础上进行二次开发，适配自定义的日志库。

* + - 1. 设计



日志框架类图

ClogRecorderBase为日志记录后端，负责真正的输入。CLogData是结构化的日志对象。ClogGuide是应用层的日志接口。CLogRecorderKeeper是为了锁定已经注入到GlobalCtrl内的日志的keeper对象（参考：全局资源管理）。

* + - * 1. 日志配置问题
    1. 字符串

1. pump\_memory库
   1. 智能内存管理框架
2. pump\_function库
3. pump\_event库
   1. 业务、请求/响应、服务、事务及事件

libpump的定位，是线程级的微服务开发框架，并提供一系列基础服务，比如网络传输，文件读写、USB通信、异步日志、智能内存管理。

首先需要明确如下概念：

* **业务**：用户业务指从提出请求->提供服务->完成请求的一系列过程。
* **请求：**用户向一个或多个服务提出的达成某种目的要求。
  + **请求实体**：用户向一个服务实体提出的具体的服务请求。
  + **请求簇：**一些列有序请求组成。比如现在有协议操作服务（组装和解析）和网络传输服务各自占用一个线程，要实现先组装协议数据，再通过网络发送到目的端。由于涉及多个服务间协作，单一请求无法完成工作。这就需要使用请求簇，请求簇包含达成用户要求的所有请求，这些请求在服务间传递，最终完成服务。
* **服务**：由服务实体提供，满足用户的某种要求的一系列活动。
  + **服务实体：**服务的具体承担者
  + **服务簇：**一组服务实体的集合
* **事务**
  + **原子事务**：一个服务由一系列原子事务组成。如网络IO服务包含发送，接收，创建连接等原子事务。
  + **事务流：**若干原子事务按一定顺序的执行流程
* **事件**：一个事务执行的启动由事件触发，而事务的执行过程又可能触发新事件。注：事件在服务内部流动，而请求在服务间流动。当服务外有请求发生时，会触发服务内部的事件。
  + 1. 请求与响应

宏观的请求指用户向一个服务簇（包含一个或多个服务实体）提出的达成某种目的要求，其功能由请求簇承担。微观的请求指宏观服务中隐含的若干的业务上不可分的要求，其功能由请求实体承担。而服务一旦完成，应向用户反馈响应结果。

* + - 1. 请求/响应实体

请求实体指用户业务逻辑上不可再分割的请求，即能够被服务实体所接受的用户业务请求，其具有原子性。



图1-1请求/响应实体

如图1-1中红色箭头表示一个请求实体，请求实体描述用户希望服务实体完成的具体服务，比如完成一次文件读写操作、完成某条http协议的组装、完成一次网络传输、完成一次日志的写入，这些都是请求实体。

与之请求实体对应的即响应实体，其描述服务实体对请求实体的响应。图1-1中蓝色箭头代表响应实体。

* + - 1. 请求/响应簇

请求簇由一系列请求实体组成，请求簇应包含实现某个用户业务逻辑的全部请求实体。请求簇是服务簇的输入，并在服务簇内部服务实体间流动。



图1-2 请求/响应簇

图1-2中导入服务簇的请求即请求簇，可能隐含一系列请求实体，红色虚线箭头表示一个请求簇。如用户的业务为：向Web服务器发送一条取数据的请求并等待响应。则请求簇应该是{ 组装http协议->发送数据->接收数据->记录日志 }，其中涉及三个服务实体，且请求簇需要规定这些服务以何种顺序和方式协作。

与请求簇对应的概念是响应簇，图1-2中蓝色虚线表征响应簇。

* + 1. 服务

宏观服务是一个抽象概念，是服务提供者向用户提供的达成其目的一系列活动，其功能由服务簇承担。而微观服务则指一个具体的服务，由服务实体承担其功能。服务簇可以由若干服务实体组成，服务实体协同工作最终实现服务。

* + - 1. 服务实体

服务实体应该要能够作为线程/进程独立运行，能够接收其他线程的服务请求。当然如果有必要，比如cpu资源紧缺时，多个服务实体也可以共同占用一个线程。

由于服务请求是异步的，服务应该有请求输入队列。服务中具体的事务流由事件启动，因此外界的请求需要被转化为服务内的事件以激活事务，处理请求。当请求处理完毕之后，需要响应用户。



图 1-3 服务实体

图 1-3中，涉及4个服务实体（黄色圆），文件操作服务、协议操作服务、网络服务及日志服务。一个完整的用户业务逻辑：用户请求一次文件读写，读写完成后通过网络向远程日志服务器写入日志。从宏观上看，4个服务整体向外界提供文件读写并记录操作日志的服务；从微观上看，四个服务实体各自提供独立的服务：文件读写、协议组织、网络传输、日志记录。

* + - 1. 服务簇

服务簇由若干服务实体组成，其可以是一个注册有多个服务实体的线程，也可以由多个服务实体线程协同组成，甚至可以由部署于不同进程的若干服务实体组成（分布式）。服务实体承担业务流程中的一个具体执行环节，而服务簇则承担一个完整的用户业务逻辑。以图1-3中的业务为例，服务簇示意图如下：



图1-4 服务簇

图1-4中四个服务实体组成一个服务组，完成一个用户业务。红色圆角矩形表示一个服务簇。服务簇只描述用户业务逻辑中所需的服务实体，而不关心服务实体间的协作。服务实体间的协作由请求簇描述。

* + 1. 事务

事务是服务实体内部的业务逻辑概念。一个服务实体一般提供一系相似的功能，比如网络服务实体，提供网络传输相关的一系列功能，比如接受连接、创建连接、接收数据、发送数据等。于是我们定义事务为服务实体中表示一类功能的实体。

* + - 1. 原子事务

原子事务是指服务实体内部业务逻辑上不可再次划分的功能。



图 1-5 原子事务

图1-5中红色圆角矩形表征原子事务。多数情况下，请求实体传达给服务实体后，服务实体需要组织执行多个事务，已完成服务。比如一次网络收发数据请求，需要{ Connect -> Send -> Recv -> Close } 4个事务参与执行，完成建立连接、发送数据、接收数据、关闭连接几个基本操作。于是引出事务流概念。

* + - 1. 事务流

事务流是服务实体内部接受外部请求后，为了完成任务发生的一系列活动。其规定了原子事务的执行顺序和方式。



图1-6 事务流

图1-6中蓝色虚线表示业务流。业务流具有方向性、顺序性。

* + 1. 事件

服务实体采用事件驱动模型，事件驱动是指在持续事务管理过程中，进行决策的一种策略，即跟随当前时间点上出现的事件，调动可用资源，执行相关任务，使不断出现的问题得以解决，防止事务堆积。

事务的执行需要事件启动，事务运行期间可能触发事件，事务结束会触发事件。事件在服务实体内部流动，使服务实体正常运行。



图1-7 事件

图1-7中绿色圈表示事件，事务流描述了服务实体一次静态的执行过程，而事件则是促使这一过程执行的驱动器。

* 1. 服务模型

服务指能够独立运行的若干服务实体线程组成的进程，而服务实体指是一个独立服务线程。一个服务实体中，可以注册若干事务，一个事务由若干原子事务组成。事务是服务功能的承担者。

由于软件规划使用插件架构开发。除了基础服务驱动，服务和事务都基于组件开发，以实现最大限度的代码复用。



图 3-1 组件、服务、消息总线关系

图3-1中示意了，Daemon服务所依赖的若干组件，包括Daemon服务组件和其依赖的若干事务组件。首先Daemon服务组件派生于基础服务组件，基础服务组件提供服务基本框架，包括通用的接口（启动/终止/挂起等），和必要的数据结构。用户从服务基类派生，实现用户自定义服务，主要是配置事务等。然后，用户从基础事务组件派生若干用户自定义事务组件如图中所示的COM管理、服务管理、消息总线管理及协议管理。

图3-1中红色箭头表示服务实体间通过*服务通信协议*交互，绿色箭头表示组件间通过*组件通信协议*交互。

组件间的通信使用码字的内部协议实现。一条组件协议对应一个原子事务，事务支持一条协议即代表事务承担协议功能。例如：用户要开发一个使用TCP协议的网络传输服务。首先定义如下组件内部协议供网络服务组件与Tcp事务组件间通信，

|  |
| --- |
| code 3-1 |
| /\*\*组件内部协议  \*#include <inter\_com\_pro.h>  \*/  struct **PRO\_Net**  {  char\* sz\_pro\_name = “PRO\_Net”;  char\* szIp; // ip地址  short iPort; // ip端口  unsigned char cProType; // 协议类型，0-TCP，1-UDP  unsigned char cOpType; // 操作类型，0-Recv，1-Send，2-Open，3-Close  Buffer buf; // io接收发送缓冲区  FunCallback cb; // io事件回调函数  }; |

然后，用户派生事务基础组件，以开发TCP事务组件。为了满足事务组件的自描述性，TCP事务组件中需要定义struct PRO\_Net{...}。并在服务组件加载事务组件时，主动向服务组件注册自己的功能。

最后，用户派生基础服务组件，开发网络服务组件。为了能和TCP事务组件通信，网络服务组件也需要定义此协议结构体。因此，网络服务组件与Tcp事务组件均依赖于组件内部协议库，又因为协议实际上就是一堆结构体，所以只需要在编译组件时包含协议的头文件即可。

当协议库扩充时，只要以前的组件用不到新增协议，就不需要重新编译之前的组件，即协议扩充不影响之前的组件。

上面的假设使用场景中，涉及依赖依赖库：*服务驱动框架*、*基础服务组件库*、*基础事务组件库*，*通信协议库*。其中服务驱动框架用于服务驱动程序开发，应该提供进/线程管理，内存管理等基础功能。基础服务组件库，则给开发者提供一套开发服务的标准和模板，包括通用的数据结构，接口等。基础事务组件库，则为开发者提供一套事务开发的模板。通信协议库则提供组件间通信协议和服务间通信协议等。

* + 1. Daemon服务

一个服务进程可以包含若干服务线程，服务进程间可以协作。因此*进程间通信管理*、*内部服务管理*、*内部消息总线管理*，需一个专门的服务实体——Daemon实现。

* + - 1. 内部服务管理

内部服务管理是Daemon服务最核心的功能。其中又包括：*服务组件管理*、*服务实体管理*两个子功能：

* 服务组件管理
* 提供服务组件配置、加载和注册等功能。组件配置提供静态和动态两种服务组件配置方式。静态配置方式使用户可以通过配置文件选择加载服务组件然后运行服务。动态配置方式则可以使Daemon服务接请求配置工作服务的参数。
* 服务组件以动态库的形式提供，用户以动态方式加载组件的动态库。
* 加载服务组件时，Daemon从服务组件获取能力注册，以确定能向用户提供的服务。
* 服务实体管理
* 提供控制服务实体生命周期的功能，包括启用/终止/挂起服务实体执行等。
* 负载均衡功能，当某类服务有若干服务实体共存时，合理分配任务提高服务处理请求的吞吐量。
* 服务进程状态管理，工作服务定期向日志写入运行状态的统计信息，Daemon维护这些统计信息，并向其他服务进程提供。
  + - 1. 内部消息总线管理

消息总线是服务实体间传递消息的通信介质。每个工作服务启动时，Daemon都会创建消息队列并绑定工作服务，最后挂载到消息总线上。当工作服务销毁时，Daemon从消息总线上卸载消息队列。

Daemon还负责向总线投递请求簇，并处理回复簇。

* + - 1. 服务进程间通信管理

服务进程间通信，目前设想采用远程过程调用的方式。涉及到协议、权限等具体问题，还需要进一步研究。

* 1. 事务模型
     1. 需求分析

事务的定义参考：[事务定义](#事务定义)。为了实现事务的复用，服务以动态库插件的形式调用事务功能。为了组件间解耦合，使用组件通信协议进行交互，即服务组件通过协议调用事务组件功能（参考：[服务组件调用事务组件功能](#服务组件调用事务组件功能)）。为了实现调用，事务组件需要定义一系列与服务组件协商好的接口和对象，主要功能包括：*自描述*、*协议管理*、*协议实现*。

* **自描述**

服务组件加载并初始化事务组件时，服务组件需要从事务组件获取组件协议版本，并检测服务组件对该版本协议的支持情况，然后获取事务组件实现的协议，并注册到服务的能力集中，当用户对服务发出请求时，服务首先需要核对能力集是否支持请求，支持才会处理，否则回复不支持。

* **协议管理**

协议管理功能，主要是接收/响应服务组件的调用请求。当接收一条服务组件的调用请求后，先根据请求类型查找对应协议的实现接口，就调用内部接口处理请求，处理完毕后返回结果（同步调用）。由于组件间通信依靠协议，因此事务组件对协议功能接口的实现形式不做要求，事务组件可对外仅提供一个协议接口，形式大致如下：

|  |
| --- |
| code 4-1 |
| /\*\*  \* @brief 事务组件对外提供的功能调用API  \* @param iCmdReq [in] 请求命令码  \* @param pProReq [in] 请求协议数据  \* @param pProRes [out] 回复协议数据  \* @return 返回执行情况，0表示成功，其他值表示执行结果状态。并结合程序错误码，可以确定执行过程中的错误。  \*/  TRANS\_API int Request(  unsigned int iCmdReq,  void\* pProReq,  void\* pProRes  ) ; |

* **协议实现**

协议的实现，即实现协议所表示功能，以code 3-1中的TCP协议为例，要实现网络IO功能，就需要从基础事务组件库，派生一个组件对象，增加接口，如下：

|  |
| --- |
| code 4-2 |
| class NetIoEDTrans  : public EvDetectorTrans  {  public:  static NetIoEDTrans \* GetInstance();  int Request(unsigned int iCmdReq, void\* pProReq, void\* pProRes);  private:  int Open(char\* szIp, short sPort); // 异步建立TCP连接，向IO事件监测器投递连接操作  int Close(int sock); // 异步关闭TCP连接，投递关闭操作  int Read(int sock, unsigned char \*szbuf, Callback cb); // 异步接收数据，投递读操作  int Write(int sock, unsigned char \*szbuf, Callback cb); // 异步发送数据，投递写操作  ...  private:  static NetIoEDTrans \* g\_instance; // 单例模式  }; |
| int NetIoEDTrans::Request(unsigned int iCmdReq, void\* pProReq, void\* pProRes)  {  switch(iCmdReq)  {  case CMD\_TCP\_OPEN:  this->Open(...)  break;  case CMD\_TCP\_CLOSE:  this->Close(...)  break;  ...  }  } |
| TRANS\_API int Request(unsigned int iCmdReq, void\* pProReq, void\* pProRes)  {  return NetIoEDTrans::GetInstance().Request(iCmdReq, pProReq, pProRes);  } |

* + 1. 详细设计
       1. 条件事件监听器

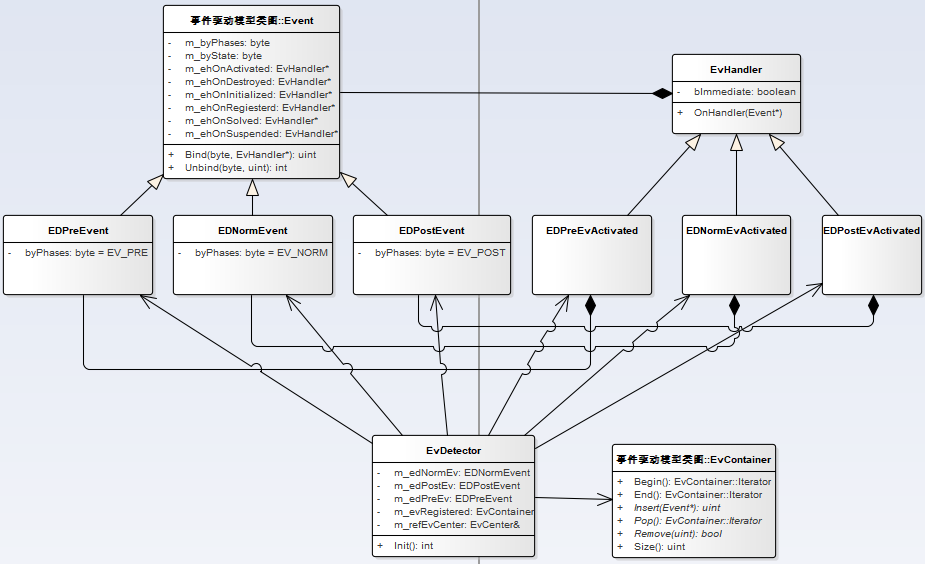


图4-1 条件事件监听器事务类图

其中，EvDetector是探测器对象，派生于事务对象。其中包含三个事件，EDPreEvent、EDNormEvent和EDPostEvent，且均属于持续性立即事件。EvDetector包含一个事件列表，用于存储已注册探测的条件事件，而探测器事务在事件循环每轮迭代中的任务就是探测这些注册条件事件是否被激活，若激活则投递到事件引擎。

* 1. 协议架构



图4-1 协议架构示意图

图4-1中，绿色箭头提示服务进程内部服务实体间通信协议，红色箭头表示服务组件与事务组件间通信协议，蓝色箭头表示服务进程间通信协议。

* 1. 事件驱动模型

服务实体为用户业务提供一些列可调用的子过程：

* 这些子过程有一定的执行次序
* 这些子过程之间需要较灵活的跳转
* 这些子过程也许需要围绕同一个上下文做操作

因此考虑使用事件驱动模型设计服务实体，以组织服务实体内部的事务。事件驱动模型中，核心对象：

* **事件Event**
* **事件句柄EventHandler**
* **事件中心 EventHub**
* **事件循环 EventLoop**
* **事件引擎EventEngine**
* **事件收集器 EventCollctor**
* **事件注册器 EventRigester**
* **事件监听器EventListener**
* **事件反应器 EventReactor**



图2-1 事件驱动模型核心，蓝色为事件引擎三大组件（注册器、监听器、反应器）、绿色为事件收集器

图2-1展示了一个事件从注册到处理的完整过程。首先，EventRigester作为事件驱动模型的使用入口，用户投递的事件插入NewlyEventMgr，此时要求对NewlyEventMgr的操作是线程安全的，需要注意一点若用户投递的事件是同步事件，插入时就会处理事件；然后在下次的事件循环中，EventRegister按照事件的类型，将事件投递到相应的事件处理器中，异步条件事件向EventListener注册，异步无条件事件向EventReactor注册；接着执行EventListener监听已注册的条件事件是否应该被激活，若应该被激活则处理事件；最后执行EventReactor，处理无条件事件。

* + 1. 需求分析1.0（预研版）
       1. Pump模型需求分析

功能描述:

Pump对象是libpump核心对象，描述事件循环的所有资源及行为。设计中，一个事件循环线程绑定一个事件循环。事件循环对象，负责管理与事件循环相关的所有对象的生命周期，包括构造、功能调用、修改、析构等。

流程图：

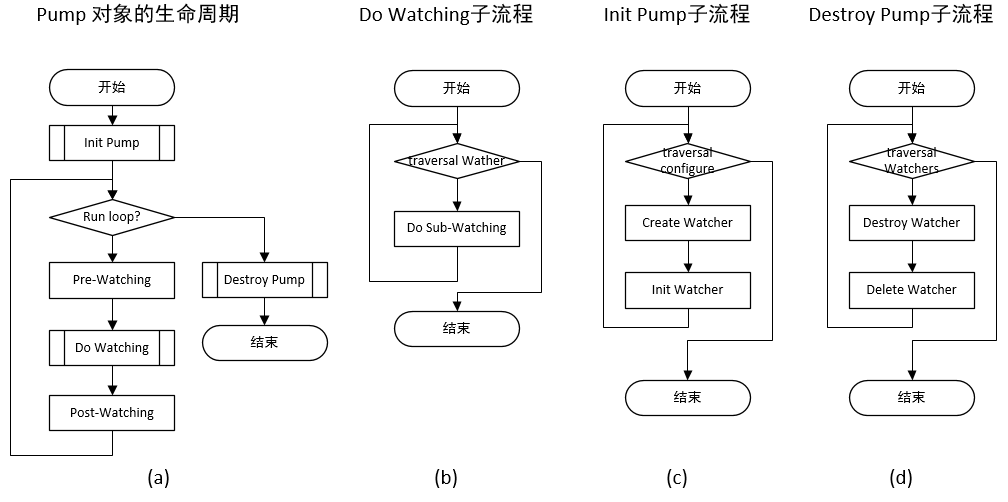


图1 (a)Pump对象生命周期流程图、(b) Do Watching子流程、(c) Init Pump子流程、(d) Destroy Pump子流程

流程说明：

启动一个事件循环线程会构造一个与之绑定Pump对象，随即开启了Pump对象的生命周期（完成构造后，Pump对象并未激活，即未开启循环，需要调用激活函数启动事件循环）。

激活Pump对象后，首先初始化Pump的资源对象，主要指按照配置要求构造以及初始化Watcher对象。Watcher对象负责监听、激活和处理关联的事件对象，不同类型的Watcher负责监听甚至处理不同类型的事件。[参考：[watcher模型需求分析](#_watcher模型需求分析)]。

进入事件循环后，每一轮循环，经历Pre-Watching、Do Watching、Post-Watching三个阶段，Pre-Watching阶段为Do Watching阶段的预处理。事件循环的核心阶段为Do Watching阶段，其功能为遍历与Pump对象绑定的Watcher对象，分别调用Watcher提供的despatch方法，检测并激活事件对象。

Post-Watching阶段做Do Watching的事后处理，比如按照优先级处理函数队列中的事件处理回调函数对象。

* + - 1. Watcher模型需求分析

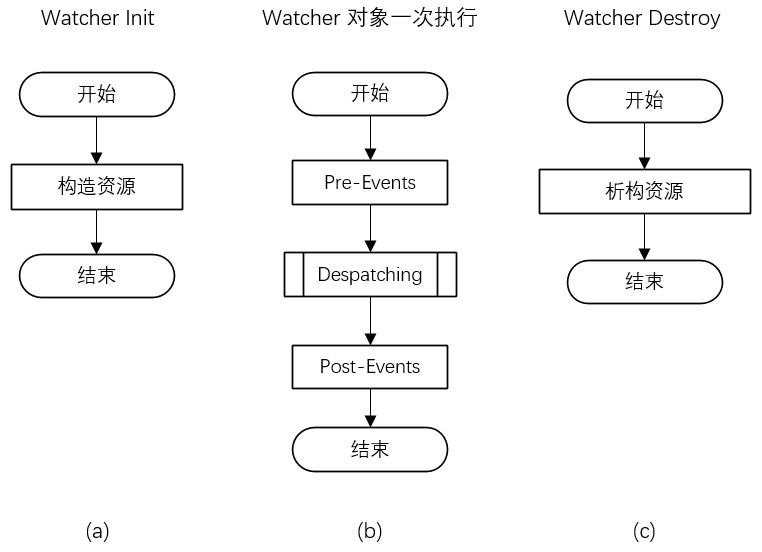
功能描述：

Watcher对象描述事件监听、激活、分发及处理功能的资源及行为。系统的执行过程中事件发生往往是异步的，软件是不可能主动知道的，需要实现某种机制去监听检测关联事件中哪些发生了，并激活（指将发生的事件单独拿出来放到某个队列中等待处理）。

不同类型的Watcher对象监听甚至包含处理不同类型的Event。比如，IO复用Watcher负责监听文件描述符IO事件、Timer Watcher监听定时器超时事件、Signal Watcher则监听其他线程的通知事件，对于普通的Watcher对象，监听到事件发生时，会根据事件的优先级将事件的处理回调函数对象发送至Pump父对象的优先级函数队列中等待统一处理。对于一些特殊的事件对象，比如流媒体处理事件，多个事件间存在时序性及依赖性（上游事件输出为下游事件输入），则要求Watcher在检测到事件发生后实时处理而不是发送一个回调函数对象等待集中处理。

Watcher之间可以相互独立，亦可以存在相互依赖的关系。比如IO-Multiplex Watcher的IO复用阻塞超时值可能依赖于先一步执行的Timer Watcher所输出的最快超时的Timer超时值。

流程图：



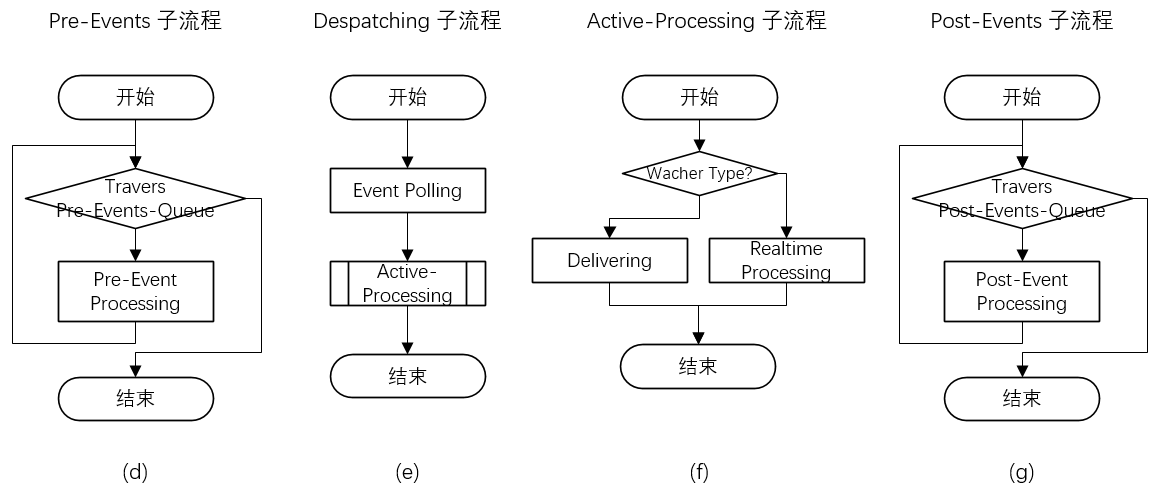


图2 (a) Watcher Init流程 (b) Watcher对象一次执行流程、(c) Watcher Destroy子流程、(d) Pre-Events子流程、(e) Despatching子流程、(f) Active-Processing子流程、(g) Post-Events子流程

流程描述：

Watcher的生命周期中，主要的功能集中在Pre-Events阶段、Despaching阶段及Post-Events阶段。其中Pre-Events执行一些已注册的预处理事件（Watcher对象应该提供Pre-Events及Post-Events的注册接口）。比如Select-Watcher或者Poll-Watcher中，需要在每次调用IO复用调用前，将需要监听的文件描述符放入列表拷贝到内核。

而最核心的功能在Despatching阶段，Despatching阶段又分为Event Polling激活事件检测子阶段和Active-Processing激活事件处理子阶段。其中：

* Event Polling主要是检测监听的事件序列中哪些事件发生，比如调用IO事件的IO复用、又或者从若干定时器中选出最早超时的定时器事件。
* 而Active-Processing激活事件处理子阶段会根据Watcher的事件处理机制有不同的处理方式。若事件间存在时序和依赖的Watcher称为Sequence-Watcher，下游事件的输入依赖于上游事件的输出，因此此事件的回调函数会在事件激活（比如信号量触发，IO事件发生等）后直接被调用。
* 将所有激活事件的回到函数对象发送至pump父对象集中处理的Watcher称为Centralized-Watcher。其处理方式类似邮件服务器的处理机制，邮件发送者将邮件发送致邮件服务器（Watcher检测到激活事件，将其回调函数对象投递至pump父对象的优先级函数队列），再由邮件服务器集中发送给邮件接收者（Pump对象集中处理回调函数队列）。

Despatching阶段完成后，还需要完成一些已注册的事后事件处理。例如对于一些要求实时性的Watcher可以在Post-Events阶段调用激活事件回调；又比如，Select-Watcher和Post-Watcher可以在Post-Events阶段交换指向三个监听fd\_set和备份fd\_set指针。

* + - * 1. Watcher——IO复用模型需求分析

Select模型使用流程的分析与设计

模型描述：

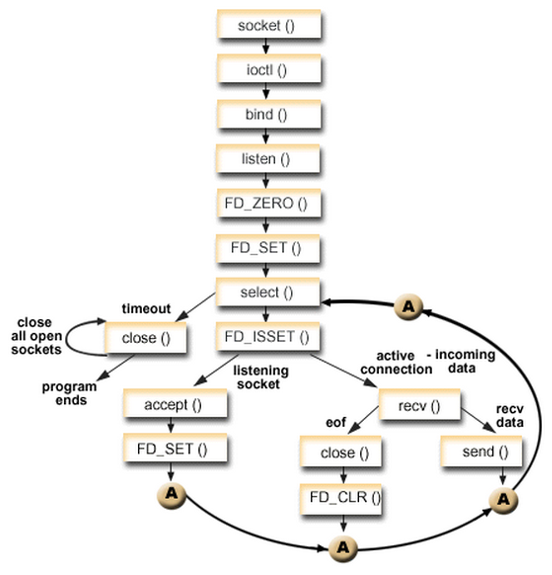


图3 select模型使用流程

如上图，Select模型的使用基本围绕fd\_set对象的操作。首先用户必须维护三种不同的fd\_set（可读、可写、可发生异常），每次调用select()前都必须将3种关联不同io事件的fd加入对应fd\_set。

这个过程最简单的处理方案就是将所有需要进行io监视的fd及其所关联的io事件组成链表，每次调用select()前，遍历链表，根据fd关联事件将对应fd\_set的位置1。但是这样做的缺点是，每次都需要遍历所有的fd，在是应用场景中一旦开始某个fd的io事件监听，在一段时间对其的监听将会持续，因此没有必要每次调用select()前都重复遍历。

优化方案：

将所有需要进行IO事件监听的fd及其关联io事件组织为select\_fd，select\_fd中存放fd以及其关注的Event对象，放入select\_fd\_list。list 逻辑结构可以是hash表，通过fd值进行索引，搜索复杂度O(1)，适用于进程管理少量fd情况（注：目前select模型中，使用的fd\_set中能够管理的fd数量是写死的1024个，即使增加进程的fd限制对其无影响，所以适合使用hash table）；或者使用平衡二叉搜索树map<fd, …>，搜索复杂度O(logN)，适用于进程管理海量fd的情况（poll模型）。然后，再建立一个fd state change队列，记录select\_fd\_list中状态发生改变的fd，比如新建了一个fd并关联事件或改变已有fd所关联的事件（如不再可写），则将fd对应的select\_fd对象放入select\_fd\_state\_change\_queue。

如此每次调用selete()前，不再遍历select\_fd\_list，而是遍历select\_fd\_state\_change\_queue，然后在上一轮的fd\_set基础上根据select\_fd\_state\_change\_queue做修改，这要求上一轮更新fd\_set时需要做备份，即使用fd\_set双缓冲的方法。本方案的优点，在于以空间换时间，减少select\_fd\_list中不必要的遍历步骤。

值得注意的一点，所有发生状态转变的fd加入select\_fd\_state\_change\_queue的同时也需要更新到select\_fd\_list，即必须保证select\_fd\_list的完整性、正确性和时效性。

流程图：

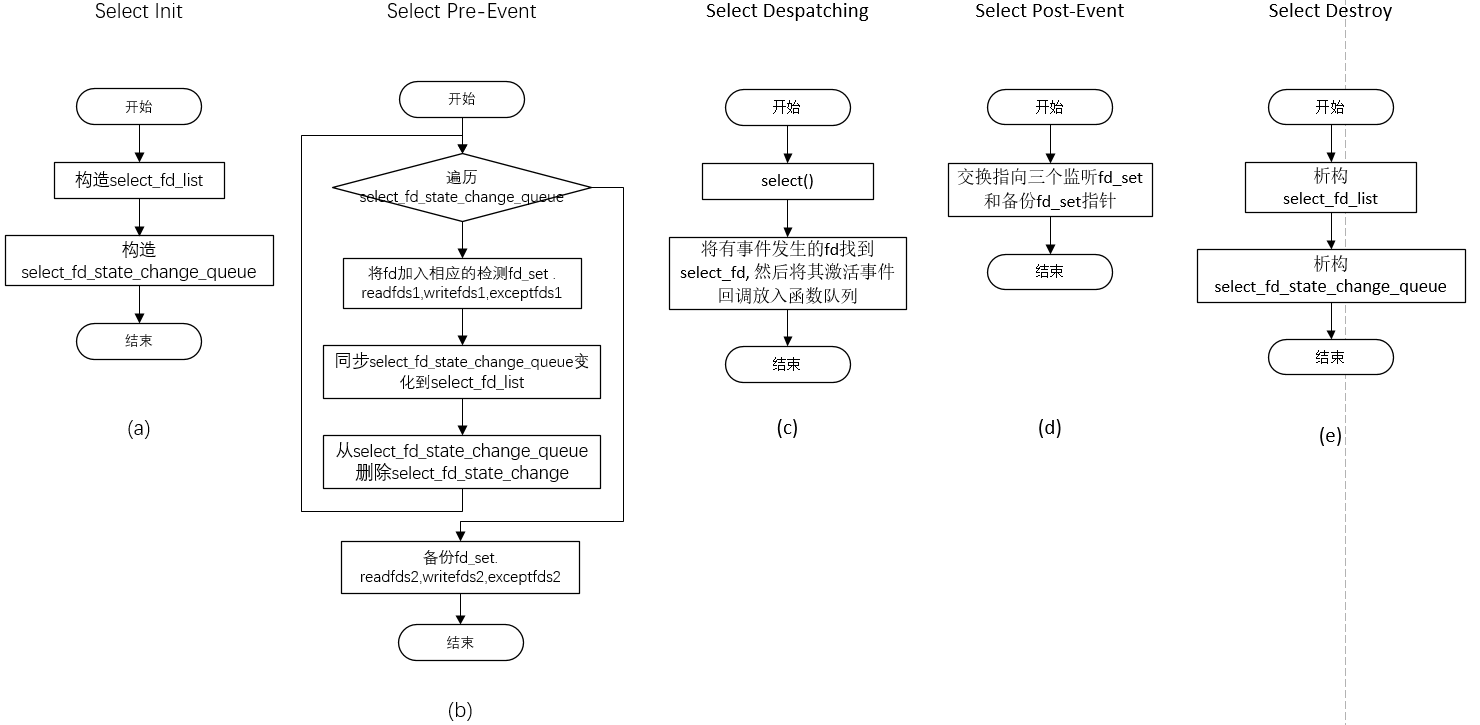


图4 Select-Watcher流程图组

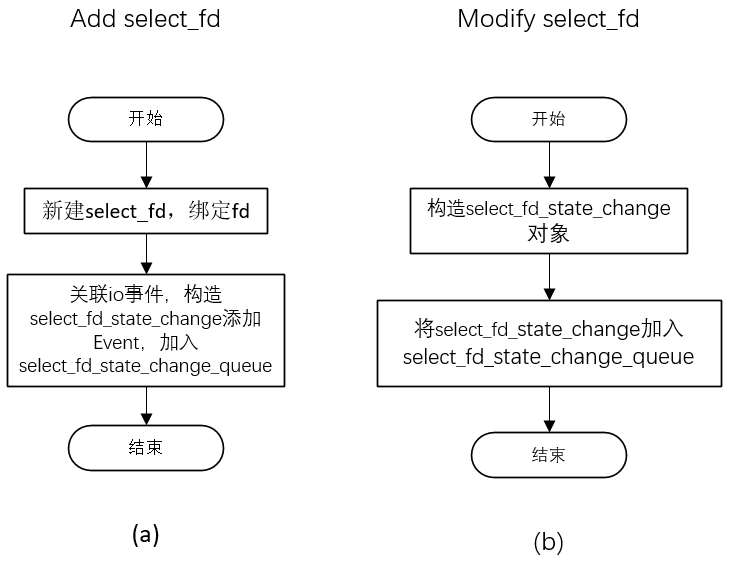


图5 (a) 新增select\_fd流程、(b) 修改select\_fd流程

修改xxx\_fd对象的请求不能直接操作xxx\_fd\_list，应为修改的请求为异步操作，即当请求发生时，xxx\_fd当前处于什么运行状态是未知的，也许此时xxx\_fd正好有需要接收的数据，如果贸然修改xxx\_fd\_list中的xxx\_fd状态，可能导致漏掉最后一次收数据的机会。因此，用户修改xxx\_fd的请求，应该以xxx\_fd\_state\_change对象的形式存放在xxx\_fd\_state\_change\_queue，再在下一轮Watching的Pre-Events阶段同步到xxx\_fd\_list。

用户提出修改请求后的处理，目前有两个可选方案：1）【同步方案】构造xxx\_fd\_state\_change对象，并直接操作xxx\_fd\_state\_change\_queue进行跟新；【异步方案】构造xxx\_fd\_state\_change对象，而将跟新到xxx\_fd\_state\_change\_queue的操作看做函数对象发送至Pump的回调邮箱。

核心对象：

* 文件描述符/句柄对象：**select\_fd**
  + io相关的文件描述符/句柄对象对象，其中封装了（以下统称）句柄相关的操作和资源。包括io句柄以及关联事件对象。
  + Select-Watcher将管理的io句柄组织为hash表或者二叉搜索树
  + select\_fd对象将io句柄关联的所有事件对象Event组织为链表，因为一个fd关联的事件对象不会太多
* 事件描述对象**select\_Event**：
  + 描述Select模型的事件对象及其处理回调函数，一个soll\_fd可以关联多个不同类型的事件（select提供可读/可写/异常三种事件类型）
* fd列表：**select\_fd\_list**
  + 存放select\_fd的列表，逻辑结构可以是hash表或者二叉搜索树
* fd状态转移队列元素对象：**select\_fd\_state\_change**
  + select\_fd状态所要发生的改变
* fd状态转移队列：**select\_fd\_state\_change\_queue**
  + 记录select\_fd状态变化的队列，select\_fd\_list应与之保持一致性

参考：[linux socket开发20180714.docx::”*6.2 I/O多路复用select模型”*]

Poll模型使用流程的分析与设计

模型描述：

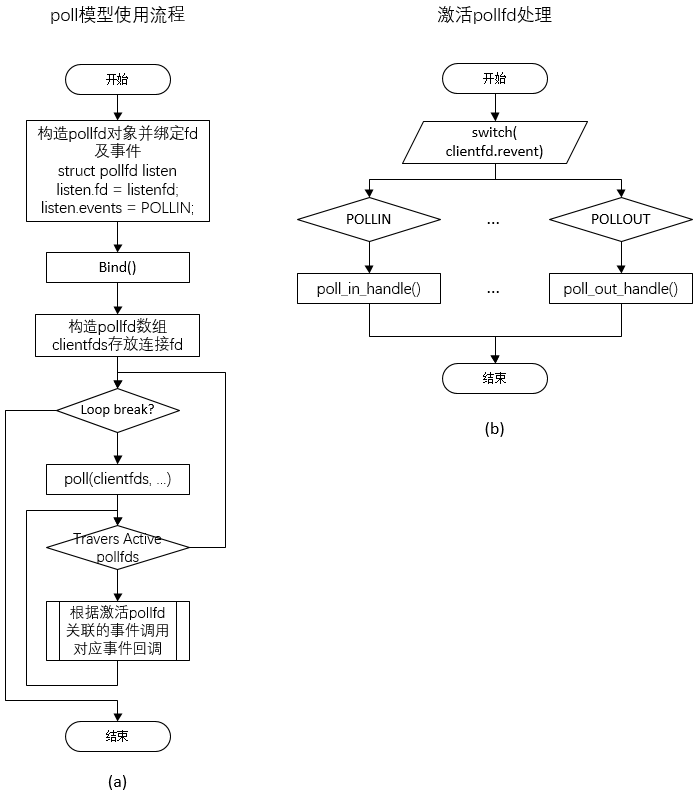


图6 poll模型使用流程

poll模型与select模型流程基本一致，最大的区别是poll模型不需要再每次调用poll()前重新构建fd\_set数组。因为poll模型提供pollfd对象：

|  |
| --- |
| struct pollfd {  int fd; /\* 文件描述符 \*/  short Events; /\* 等待的事件 \*/  short rEvents; /\* 实际发生了的事件 \*/  } ; |

用于存放文件描述符及其关联的io事件，且增加一个成员rEvents用于存放poll返回后实际发生的事件。这意味着使用者不需要再在每次调用io复用前遍历所有fd并根据其关联的事件添加到相应的fd\_set中。

另外Poll为fd提供的io事件划分更细，可处理的行为更多。

流程图：

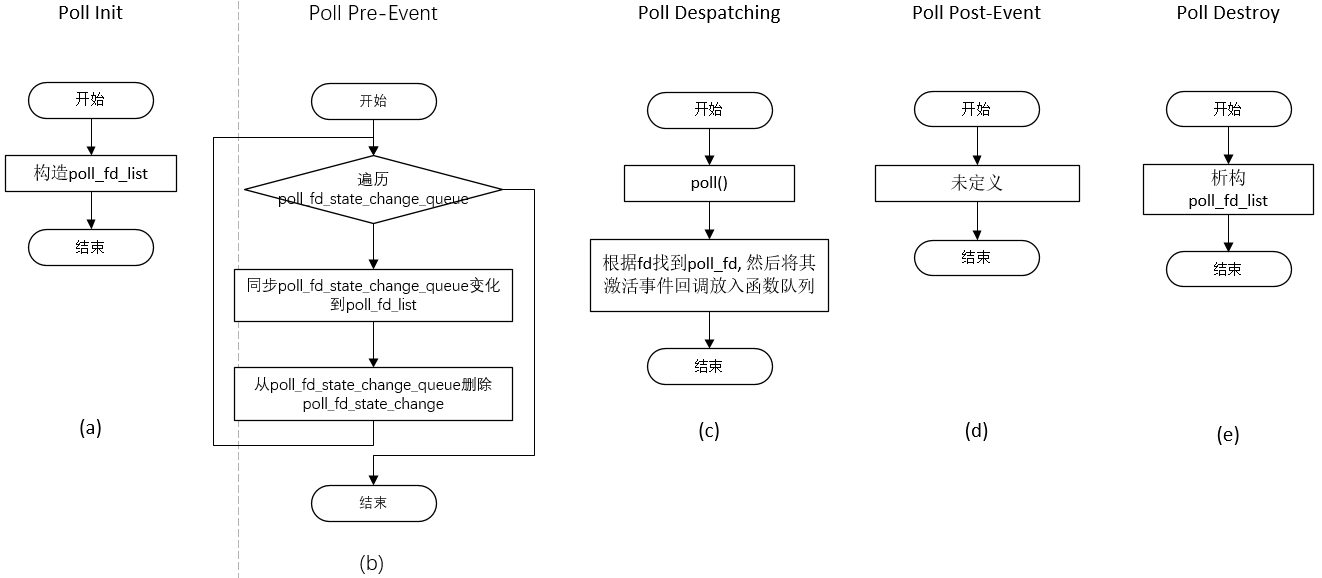


图7 Poll-Watcher流程图组

对比Select-Watcher和Poll-Watcher的流程图组，发现二者真正不同之处只是在xxx\_fd对象的维护处理上存在差异，io复用的基本框架都是一致的。

核心对象：

* 文件描述符对象**poll\_fd**：
  + poll\_fd直接使用操作系统提供的poll IO复用机制的pollfd对象，包括绑定io fd、关联事件及激活事件三个成员。
* 事件描述对象**poll\_Event**：
  + 描述poll模型的事件对象及其处理回调函数，一个poll\_fd可以关联多个不同类型的事件。poll支持事件类型，参考  
    <https://blog.csdn.net/q8250356/article/details/81058478>
* 文件描述符队列**poll\_fd\_list**：
  + poll\_fd数组，封装了操作poll\_fd数组的操作，类似于select\_fd\_list
* fd状态转移队列元素对象：**poll\_fd\_state\_change**
  + poll\_fd状态所要发生的改变
* fd状态转移队列：**poll\_fd\_state\_change\_queue**
  + 记录poll\_fd状态变化的队列，poll\_fd\_list应与之保持一致性

Epoll模型使用流程分析及设计

模型描述：

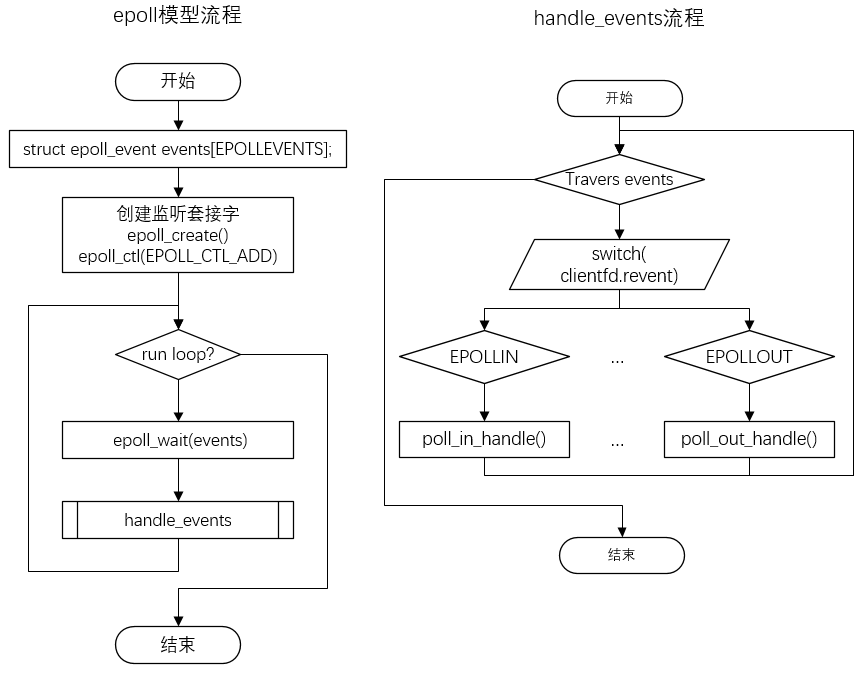


图8 epoll模型流程图

epoll模型与poll和select模型在运行原理上基本相似，但其在性能上有很大改进，因为epoll模型消除了每次调用epoll()时将监听xxx\_fds对象拷贝到内核这一步骤。fds的维护工作由内核完成，使用者只需在添加或修改fd时调用epoll\_ctl()方法通知内核一次即可。与poll和select模型相比，epoll模型提供了更为复杂的使用接口和数据对象。

|  |
| --- |
| epoll操作过程需要三个接口 |
| #include <sys/epoll.h>  int epoll\_create(int size); // 构造epoll句柄  int epoll\_ctl(int epfd, int op, int fd, struct epoll\_Event \*Event); // 添加或修改fd状态  int epoll\_wait(int epfd, struct epoll\_Event \* Events, int maxEvents, int timeout); // io复用检测 |
| 核心数据结构 |
| // epoll事件对象  struct epoll\_Event {  \_\_uint32\_t Events; /\* Epoll Events \*/  epoll\_data\_t data; /\* User data variable \*/  };  // epoll用户数据对象  typedef union epoll\_data {  void\* ptr;  int fd;  uint32\_t u32;  uint64\_t u64;  } epoll\_data\_t; |

与select及poll不同，调用epoll\_wait()返回的对象时epoll\_Event数组而不是xxx\_fds，这意味着用户只知道发生么什么事件，而事件发生在哪个fd上，则需要由epoll\_data对象指明。可以看到，epoll\_data是一个联合体，除了可以存放fd之外，还可以存放void\*指针，这意味着用户可以用epoll\_data夹带私有数据。比如，在即时通讯程序中，将用户账号和fd绑定以后放在结构体struct user{int fd; char \* name;}中，则可以将epoll\_Event.data.ptr = (struct user\*)\_user;是fd与用户名绑定。这样，当fd收到数据时就知道是哪个用户收到数据了。

流程图：

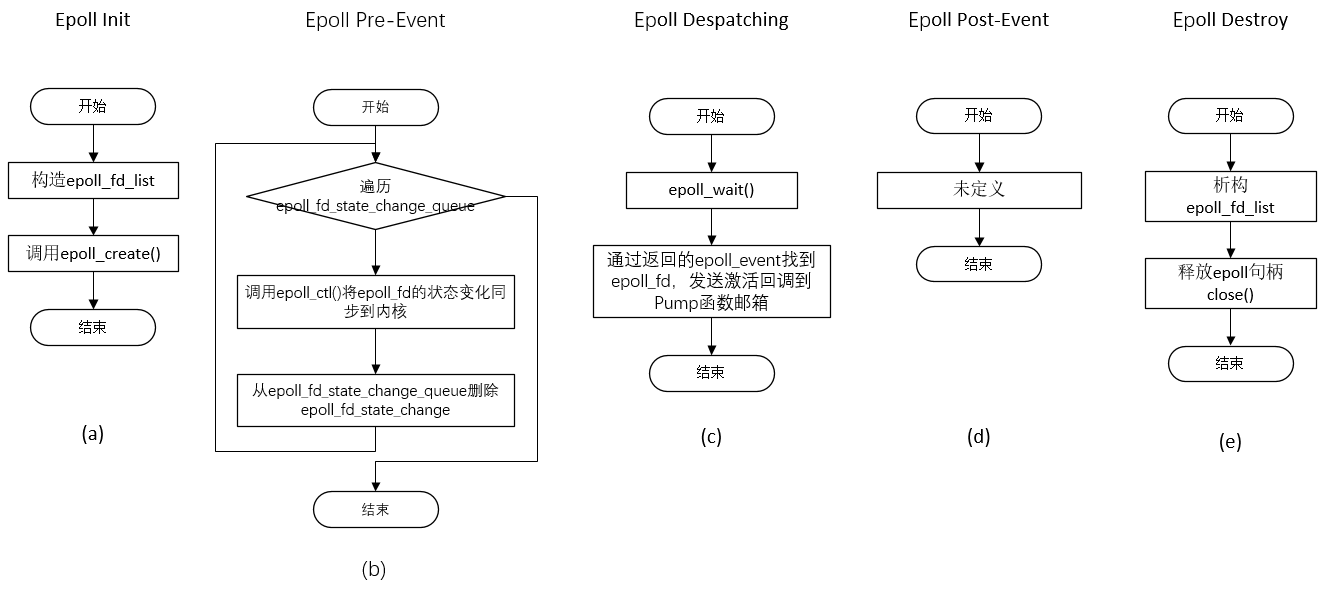


图9 Epoll流程图组

Epoll流程与Poll及Select相比，还有一个不同之处在于，Epoll本身占用一个文件描述符资源，使用epoll\_create()创建。所以在销毁Epoll对象时需要关闭epoll文件描述符。

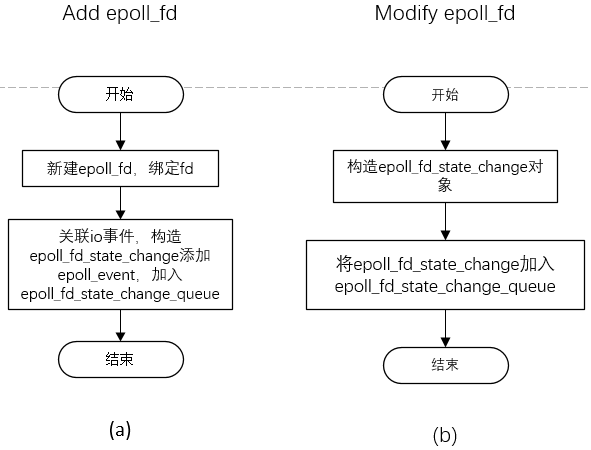


图10 Epoll中epoll\_fd的管理流程

Epoll模型中对于fd的管理模式与Poll及Select相同，即使用xxx\_fd\_state\_change代理状态修改请求，并在下一轮循环Pre-Events同步到xxx\_fd\_list。区别是除了同步时还需要调用epoll\_ctl将变化同步到内核。

核心对象：

* 文件描述符对象**epoll\_fd**：
  + epoll\_fd直接使用操作系统提供的epoll IO复用机制的epollfd对象，包括绑定io fd、关联io事件及激活事件三个成员。
* 事件描述对象**epoll\_Event**：
  + 描述epoll模型的事件对象及其处理回调函数，一个epoll\_fd可以关联多个不同类型的事件。epoll支持事件类型，参考  
    <https://blog.csdn.net/q8250356/article/details/81089920>
* 文件描述符队列**epoll\_fd\_list**：
  + epoll\_fd数组，封装了操作epoll\_fd数组的操作，类似于select\_fd\_list
* fd状态转移队列元素对象：**epoll\_fd\_state\_change**
  + epoll\_fd状态所要发生的改变
* fd状态转移队列：**epoll\_fd\_state\_change\_queue**
  + 记录epoll\_fd状态变化的队列，epoll\_fd\_list应与之保持一致性
    1. 需求分析2.0（决定版）

Event事件对象是事件驱动模型的核心对象，代表系统或软件中发生的某种变化，并描述用户对这种变化的期望响应（事件句柄）和产生响应的必要资源（数据）。如一次文件IO，等待系统通知，一次网络通讯，等待定时器都看做一个事件。事件有6种基本生命周期状态。事件状态发生改变，比如文件IO完成、期待的系统通知抵达、一次网络IO完成、定时器超时，都会引起正在监听的事件激活。

事件间可能存在某种关联，比如用户要求读取10秒钟的指定文件，在这个事件组中涉及两种事件：读文件事件和定时器事件。首先创建一个文件读事件，并在成功注册事件检测器时调用监视器注册回调，创建并注册定时器事件，接下来不断处理激活的文件读写事件，当在定时器超即定时器事件激活时，调用定时器的事件句柄关闭文件句柄，并从事件监听器中注销文件读写事件。



图2-2 读取10秒钟指定文件的事件驱动模型处理示意图

事件的生命周期，分为：

* EVSTATE\_NEWED = 0, //! 新建态,事件加入到NewlyEventMgr时
* EVSTATE\_LISTENED, //! 监听态,新建态条件事件注册到ListenedEventMgr
* EVSTATE\_ACTIVATED, //! 激活态,立即事件注册到ActivatedEventMgr
* EVSTATE\_SUSPEND, //! 挂起态,持续性事件可以暂时被挂起不执行
* EVSTATE\_SOLVED, //! 已处理态
* EVSTATE\_DESTROYED //! 销毁态

每种状态在达成时都调用对应的状态转移句柄（state transition handler）。比如上一个应用场景中，则可以在文件读写事件成功注册事件检测器时调用状态转移函数，创建并注册定时器事件，最后在激活态状态转移句柄中销毁文件IO事件。

* + - 1. 流程说明
         1. 事件模型执行流程



图2-3 事件模型执行流程图

其中，初始化步骤完成事件模型资源初始化，如构造事件收集器、投放固定周期性事件（如fd事件监听器）；启动事件循环后，每一轮循环，事件模型的主要工作就是遍历事件搜集器是否有激活事件并逐个处理；直到结束事件循环，执行销毁步骤，释放事件模型资源。

* + - * 1. 事件循环创建流程



图2-4 事件循环创建流程

事件引擎每轮循环最主要的工作是处理事件收集器中的激活事件。处理的顺序按照PreEvent->Event->PostEvent的顺序进行处理。一般情况下，周期性的事件，如定时器超时事件、fd探测器事件等在事件中优先级较高的周期性事件放在PreEvent列表中做优先处理；普通的事件则放入Event列表中处理；优先级较低的周期性事件放入PostEvent列表中处理，如select模型每轮检测完毕后需要刷新fd列表，则可以在PostEvent列表中完成。

* + 1. 概要设计
       1. 模块说明

|  |  |  |
| --- | --- | --- |
| 编号 | 模块名 | 状态 |
| PD001 | Event | 新增 |
| PD002 | EventLoop | 新增 |
| PD003 | EventEngine | 新增 |
| PD004 | EventCollector | 新增 |
| PD005 | EventRigester | 新增 |
| PD006 | EventListener | 新增 |
| PD007 | EventReactor | 新增 |

* + - 1. PD001事件Event

事件Event对象，描述一个事件的行为和数据，事件可以看作一系列闭包的集合。

为了能够统一处理异常。以及针对异常打出日志，除了业务相关的事件，还增加了异常事件和日志事件。当然相应的也应该新增与之对应的事件句柄。

* + - * 1. 事件源

目前已知的事件，来源基本分为两类：

* **OS事件**
* **signal事件**<被动检测>

当os发送一个signal信号，程序会直接中断转去处理信号处理函数

* **fd事件**<主动检测>

需要使用select等fd复用系统调用去检测fd有事件发生。fd事件比较特殊，事件的发生是异步的，比如网络数据抵达，或者网络数据发送就绪，这些事件的发生需要通过fd事件监听器检测。而为了让fd事件监听器覆盖某一个fd，又必须向fd事件监听器注册该fd事件。

* **用户事件**<主动检测>
* 按执行时机划分：
* 同步事件

同步时事件在投递事件的线程中处理，并立即返回处理结果

* 异步事件

异步事件的投递和处理操作可以不在同一个线程中完成，事件请求线程将事件投递到事件引擎绑定的线程事件缓冲区，投递函数即可返回。而事件的处理结果需要在回调函数中得到。

* 按执行周期划分：
* 持续性事件

持续性事件处理完毕后应立即向事件引擎注册自己（事件发生，或称激活）。比如，周期定时器的触发实际涉及两个事件：fd事件、io复用检测事件。首先需要创建timerfd对象（立即事件），然后需要向*io复用检测事件句柄*注册该fd事件，每次io复用检测事件发生时都会检测定时器超时情况，超时后，激活定时器事件并处理，处理完后再投递一个定时器。

* 一次性事件

与持续性事件相反，事件不会重复触发，只会触发一次，事件被激活并处理后即销毁。

* 按执行条件划分：
* 条件事件

满足某种条件后才会激活的事件，比如定时器事件，fd事件也可以认为是条件事件。因为条件事件的发生需要满足某种条件，因此必须有*条件检测器*，fd事件监听器就是一种条件检测器，同时条件检测器也是事件句柄，为了让其长期运行，我们将条件检测器的运行注册为持续事件。

* 立即事件

直接向事件引擎注册一个激活事件，无需满足条件即可得到处理，等价于一个异步调用。

* + - * 1. 事件生命周期状态机

事件的生命周期，应该具有创建、监听、激活、挂起、处理、销毁6种基本状态。



图2-5事件生命周期状态转换

图2-5中展示了事件生命周期状态的转换。事件的初始化的概念不同事件触发（或称激活）。事件初始化指设定事件的必要参数，比如事件的类型（持续性、一次性、立即、条件事件）、需要注册的条件检测器、激活条件、激活后调用的事件句柄等。

同步事件，由投递事件的线程自行处理返回，事件不会进入事件循环对应的线程；异步事件则需要投递到事件循环的NewlyEventMgr，然后等下个循环周期注册，进而处理。

若事件属于立即事件，则直接标记为激活态，并投递到事件反应器，等待下一个事件循环周期处理。

若事件属于条件事件则需要先注册事件到条件检测器，此时事件变为监听态。

条件事件注册后，若事件发生，则状态从监听态转变为激活态，并由监听器调用激活处理函数；

而如果用户希望暂时停止某个注册的持续性事件，则转为挂起态；

同步事件和一次性事件是不允许挂起的，且这两种类型事件的事件句柄无意义，因为系统会在处理完事件自动归还句柄。

激活事件处理后转为完成态。处于完成态的事件可以恢复到初始态，重复生命周期，也可以进入销毁态进而结束生命周期。持续性事件句柄由用户自行释放。

* + - * 1. 条件事件

条件事件的激活处理需要满足一定的条件。比如Fd读事件，需要有数据抵达才会激活。因此，需要由事件监听器来监听，事件是否应该激活。不同类型的条件事件，因触发事件的条件机制不同，需要相应类型的监听器来监听。因此条件事件和事件监听器是N-1的对应关系。

* + - * 1. 事件句柄 EventHandler

事件句柄与事件是多对一关系，事件激活后，事件句柄用来执行用户的业务逻辑。

* + - * 1. 事件容器EventContainer

事件容器，要求能以任何逻辑存储结构缓存事件对象，并能够提供线程安全的使用方式。

* + - 1. PD002事件循环 EventLoop

事件循环是事件引擎的驱动，这个对象包含线程相关的操作，让事件引擎运转起来。

* + - 1. PD004事件引擎 EventEngine

当事件激活时，事件引擎找到该事件的句柄，逐个调用他们。将各子模块的执行集中在这里管理，还有两个额外的好处：

1 如果发生异常，则呼起异常处理器。这样，一旦业务模块发生了不得不终止整个过程的时候，不需要自己写try/catch子句，而只需要将异常往上抛，直到抛给框架层，由它来做这些统一的事情。然而这并不意味着各业务模块彻彻底底地摆脱了难看的try/catch/finally，运行时发生的异常被catch后，并非都可以直接END，何去何从仍然视情况而定，直接将异常吞掉也未尝不可能。

2 打日志的活儿交给EventEngine就好了，没人比它更清楚当前执行到了哪一步。而各子模块里面，可以省去许多散布在各处的日志语句。对于散弹式日志的问题，解决方法不止一种，AOP也是个不错的选择。

* + - 1. PD004事件收集器 EventCollector

事件容器的管理器，负责管理事件容器，注意不负责构造和销毁事件容器，只负责管理，属于半托管。事件容器起到事件缓存的作用，在线程间传递和缓存事件对象。事件容器由与容器关联的事件监听器创建并注册到与事件引擎绑定的事件收集器中，并在事件监听器销毁时销毁事件容器。

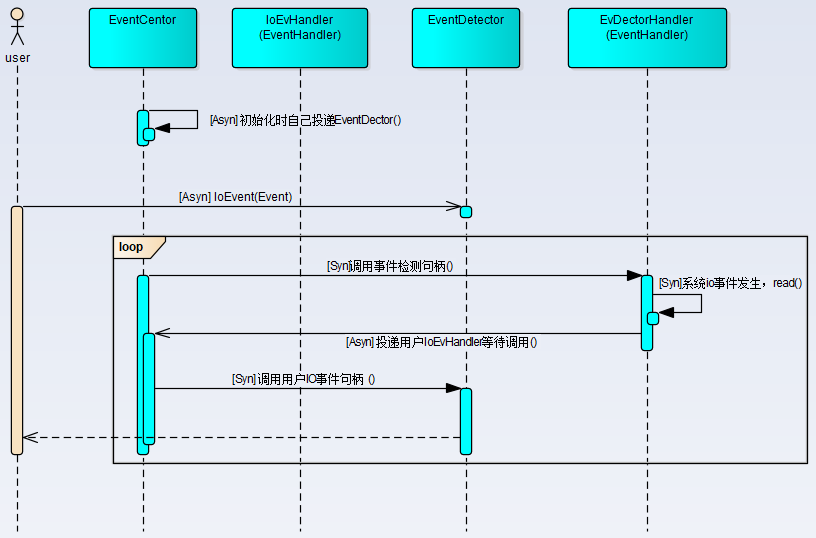
* + - 1. PD005事件注册器 EventRigester

异步事件需要等待合适的事件循环的执行周期，才能被处理。因此客户请求的处理的事件需要先存放在NewlyEventMgr中，然后由注册器注册到对应的处理器中，等待处理。条件事件被注册到EventListener中，无条件事件则注册到EventReactor中等待处理。

* + - 1. PD006事件监听器 EventListener

事件监听器功能为检测条件事件是否应该被激活，比如某个被监听的文件描述符是否有系统io事件发生，如果可读则读入数据，并激活用户io事件。

此处关于IO事件处理方案讨论：当事件监听器检测到系统io事件后，  
 1）直接读取数据到用户io事件的数据缓冲区，然后激活用户io事件，再由事件引擎调用用户io事件句柄作进一步处理；  
 2）直接激活用户io事件，由事件引擎调用用户io事件句柄从文件描述符接收数据到接收缓冲区，再做进一步处理。



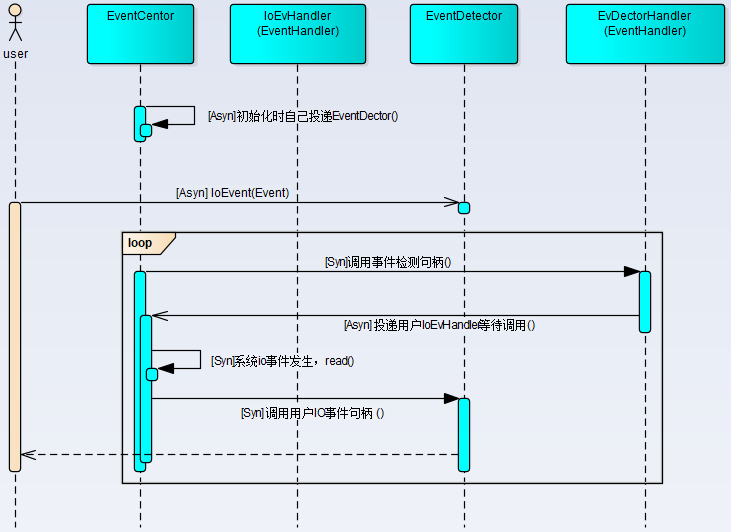


图 2-6 两种事务时序图，（上）事务逻辑1），（下）为事务逻辑2）

其中，2-6（上）时序真正的io操作发生在EvDectorHandler中，而（下）时序中发生在IoEvHandler中。相比之下（上）时序中，对于系统io事件响应更及时，当io密集时不容易丢包，吞吐率高。

* + - 1. PD007事件反应器 EventReactor

事件反应器的作用和地位与EventListener相似，只不过，其处理立即（一次性）事件。

* + 1. 详细设计
       1. 系统模块层次

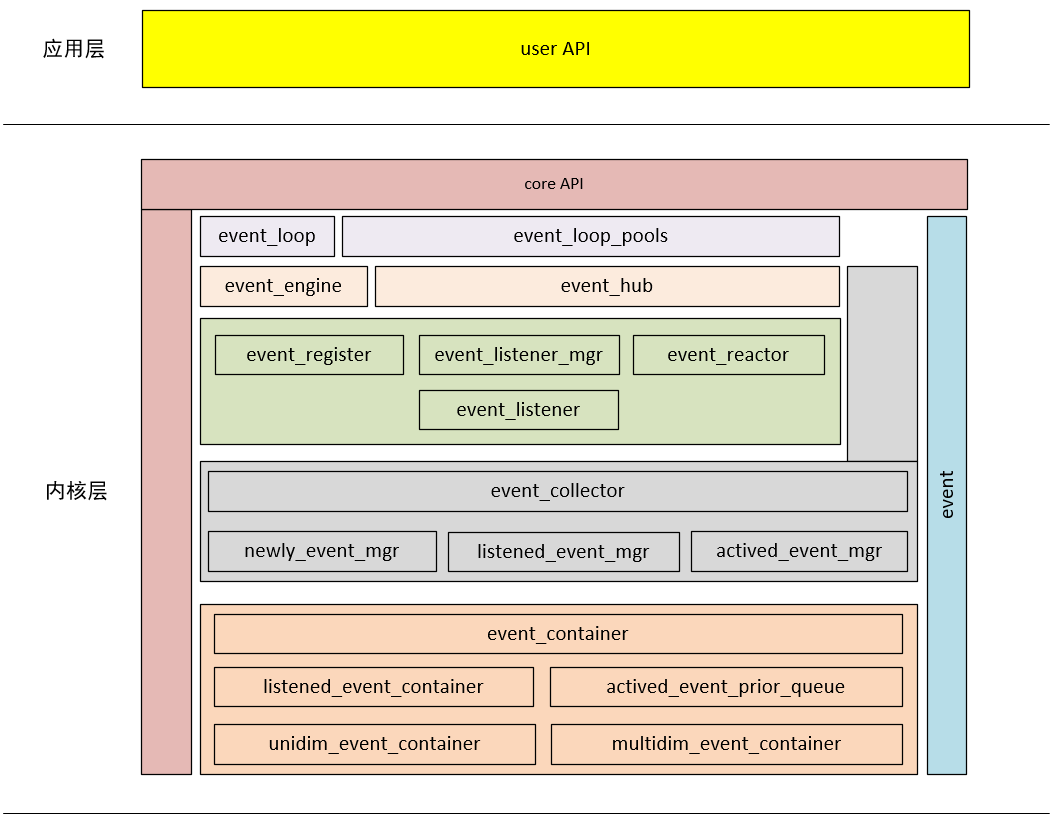


图 各模块层次关系

上图中，依赖关系大体上遵循自上而下，从左到右的顺序。core API贯穿整个事件驱动模型的内核层，起到承上启下的作用。依赖的底层若需要使用上层的对象的功能，需要通过调用core API。



系统数据流图

其中方框为静态对象（类对象），圆圈为动态对象（线程），蓝色箭头为数据流的方向，黄色箭头为调用流方向。

* + - 1. DD-001事件模块

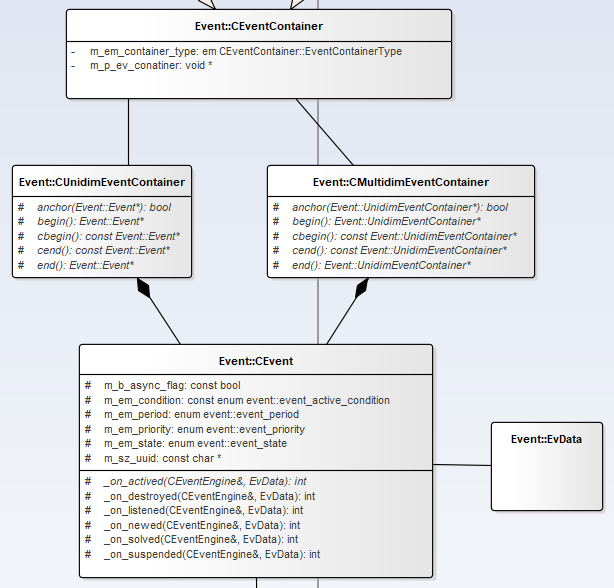


图2-7 事件对象类图

其中，Event为事件对象基类，所有事件都从此类派生。Event虽然非抽象类，但是禁止直接构造，只允许其派生类构造。另外，Event--EventListener是相互绑定的（通过uuid），Event只能注册到相同uuid的listener中。

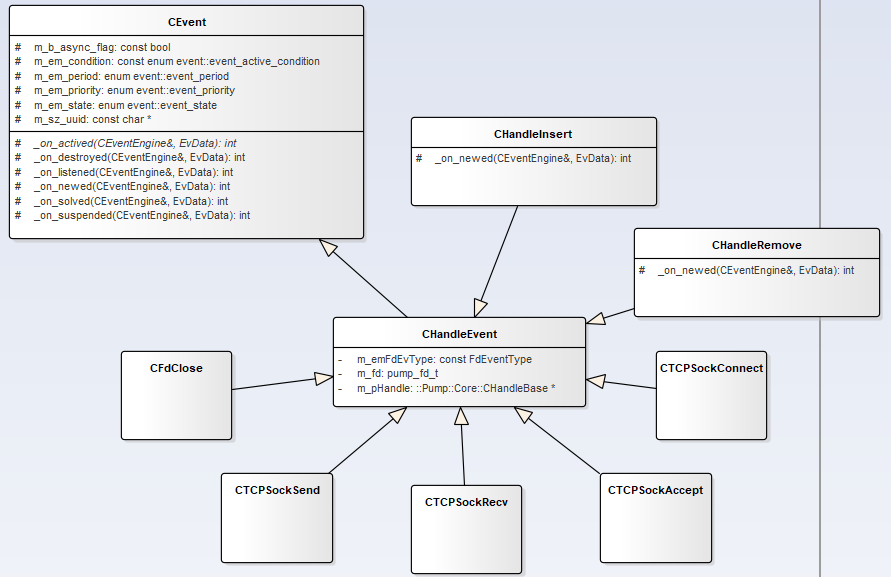


图2-8 CEvent及其派生类

其中，CHandleEvent是所有句柄事件的基类。注意成员m\_pHandle，为pump\_core的CHandleBase，这个对象时所有系统句柄（file\socket\pipe\timer等）的抽象。

* + - * 1. DD001-001 CEvent

pump\_event最基础的类，所有事件的基类。没有设计成抽象类，是应为并不是每一个派生事件都需要重载所有状态转换回调。

本质上CEvent对象是一个状态机（目前未从pump\_core@CStateMachine派生）。

成员：

|  |  |
| --- | --- |
| const enum **EventPriority** m\_emEvPriority | 事件的优先级，决定时间激活后被处理的优先级 |
| const **bool** m\_bAsync | 同步事件或异步事件，同步事件由产生事件的线程自行处理，异步事件由事件驱动模型处理 |
| enum **EventState** m\_emEvState | 事件的生命状态 |
| enum **PeriodType** m\_emEvPeriod | 事件触发周期属性<可更改> |
| const enum **ConditionType** m\_emEvCondition | 事件触发条件属性<不可更改> |
| const **char** \*m\_szUuid | 事件的资源号, 与CEventListener配对 |
| ~~EvData m\_evData[6]~~ | ~~事件状态转换回调用户数据，每一个回调对应一个data~~ |

方法：

|  |  |
| --- | --- |
| virtual int **\_on\_newed**(CEventEngine &, EvData) | 事件加入CEventCollector后回调，同步事件必须重载 |
| virtual int **\_on\_listend**(CEventEngine &, EvData) | CEventRegister注册事件时回调，异步条件事件必须需要重载 |
| virtual int **\_on\_actived**(CEventEngine &, EvData) | 事件激活时调用，即事件句柄 |
| virtual int **\_on\_suspended**(CEventEngine &, EvData) | 事件被挂起时调用，异步条件事件按需重载 |
| virtual int **\_on\_solved**(CEventEngine &, EvData) | 事件处理完一次激活后调用，可以看作\_on\_actived的后续 |
| virtual int **\_on\_destroyed**(CEventEngine &, EvData) | 事件销毁前调用 |
| ~~void~~ **~~set\_data~~**~~(EventState emEvState, EvData objData)~~ | ~~设置指定状态转换回调的用户数据~~ |
| ~~EvData &~~**~~get\_data~~**~~(EventState emEvState)~~ | ~~获取指定状态转换回调的用户数据~~ |

* + - * 1. DD001-002 CHandleEvent

CHandleEvent是文件描述符相关事件的基类。如果需要新增的文件描述符事件需要增加事件类型，比如定时器需要新增创建事件、启动事件

成员：

|  |
| --- |
| /\*\*  \* io文件描述符  \*/  ::Pump::Core::CHandleBase \* **m\_pHandle** |
| /\*\*  \* @var unsigned short re\_fd\_ev\_  \* @brief io文件描述符事件fd事件类型  \* - FDEV\_TYPE\_ACCEPT  \* - FDEV\_TYPE\_OPEN  \* - FDEV\_TYPE\_READ  \* - FDEV\_TYPE\_WRITE  \* - FDEV\_TYPE\_CLOSE  \* - FDEV\_TYPE\_SHUTDOWN  \* …  \*/  const FdEventType **m\_emFdEvType** |

方法：

|  |
| --- |
| /\*\*  \*  \* @return fd文件描述符的系统监听事件, 可取或(非互斥)。  \* - PUMP\_IOEV\_IN  \* - PUMP\_IOEV\_OUT  \* - PUMP\_IOEV\_ERR  \*/  unsigned short **get\_io\_event\_type**() |

DD001-00**2-00**1 CHandleInsert

ChandleInsert事件将文件事件加入到句柄事件容器，CHandleEventListener:: m\_pEvContainer。这是一个同步事件。

|  |
| --- |
| /\*\*将文件事件加入到句柄容器中，linux为hash表，windows为map表\*/  virtual int **\_on\_newed**(CEventEngine &refEvEngine, EvData objData); |

注：ChandleInsert、CHandleRemove两个事件主要是把一个句柄对象加入到句柄容器。加入句柄容器是为了方便多路复用器检测。

DD001-002-002 CHandleRemove

|  |
| --- |
| /\*\*将文件事件从句柄容器中删除\*/  virtual int **\_on\_newed**(CEventEngine &refEvEngine, EvData objData); |

DD001-002-003 CTCPSockConnect

DD001-002-004 CTCPSockAccept

成员：

|  |
| --- |
| /\*\* accept参数\*/  SockAcceptArg m\_struArg; |
| /\*\*用户回调\*/  OnAcceptCbType m\_cbAccept; |

方法：

|  |
| --- |
| /\*\*用户回调\*/  typedef ::Pump::Function::CFunction<  PUMP\_MFN\_TYPE(CTCPSockAcceptHandler,  pump\_int32\_t,  (::Pump::Core::Net::CSock \*, CHandleCap\*, pump\_pvoid\_t))  > **OnAcceptCbType**; |
| /\*\*绑定用户回调\*/  void **Bind**(OnAcceptCbType & cb); |
| /\*\*  \* @fn \_on\_listend  \* @param refEvEngine 事件驱动引擎  \* @param objData  \* @return 状态码  \* @brief 启动套接字监听, 向 Backend 添加IO事件  \* 1 并获取 fd\_event\_listener 绑定的事件容器  \* 2 并获取文件描述符对象FDescriptor  \* 3 并获取 fd\_event\_listener  \* 4 跟新Backend事件  \* 5 更新文件描述符对象状态  \*/  virtual int **\_on\_listend**(CEventEngine &refEvEngine, EvData objData); |
| /\*\*  \* @fn \_on\_actived  \* @param refEvEngine  \* @param objData  \* @return 状态码  \* @brief 有连接请求, 调用accept(), 调用用户回调  \* 1并获取 fd\_event\_listener 绑定的句柄容器  \* 2 accept  \* 3更新 backend Io事件  \*/  virtual int **\_on\_actived**(CEventEngine &refEvEngine, EvData objData); |
| /\*\*  \* @fn \_on\_destroyed  \* @param eventCollector  \* @param objData  \* @return 状态码  \* @brief 销毁套接字  \* 1. 并获取 fd\_event\_listener 绑定的文件句柄容器  \* 2. 并获取文件描述符对象FDescriptor  \* 3. 并获取 fd\_event\_listener  \* 4. 从多路复用对象删除fd  \*/  virtual int **\_on\_destroyed**(CEventEngine &refEvEngine, EvData objData); |

DD001-002-005 CTCPSockRecv

成员：

|  |
| --- |
| /\*\* 接收缓冲区\*/  ::Pump::Memory::IoBuffer m\_objIoBufRecv; |
| /\*\*用户回调\*/  OnRecvCbType m\_cbRecv; |

方法：

|  |
| --- |
| /\*\*用户回调对象\*/  typedef ::Pump::Function::CFunction<  PUMP\_MFN\_TYPE(CTCPSockRecvHandler,  pump\_int32\_t,  (pump\_pvoid\_t))  > OnRecvCbType; |
| /\*\*绑定用户回调\*/  void Bind(OnAcceptCbType & cb); |
| /\*\*  \* @fn \_on\_actived  \* @param refEvEngine  \* @param objData  \* @return 状态码  \* @brief TCP连接有数据到达  \* 1并获取 fd\_event\_listener 绑定的句柄容器  \* 2并获取文件描述符对象FDescriptor  \* 3 recv data  \* 4 if 出错: 关闭  \* 5 elif 正常: 处理数据  \*/  virtual int \_on\_actived(CEventEngine &refEvEngine, EvData objData); |
| /\*\*套接字关闭处理\*/  pump\_int32\_t \_\_OnRemoteClose(CHandleCap \* pFDescriptor, CEventEngine &refEvEngine); |
| /\*\*套接字接收数据处理\*/  pump\_int32\_t \_\_OnRecv(); |

DD001-002-006 CTCPSockSend

成员：

|  |
| --- |
| /\*\*发送缓冲区\*/  ::Pump::Memory::IoBuffer **m\_objIoBufSend** |
|  |

方法：

|  |
| --- |
| /\*\*  \* @fn \_on\_actived  \* @param refEvEngine  \* @param objData  \* @return 状态码  \* @brief TCP连接数据发出去  \*/  virtual int \_on\_actived(CEventEngine &refEvEngine, EvData objData); |

* + - 1. DD-002 事件驱动模块

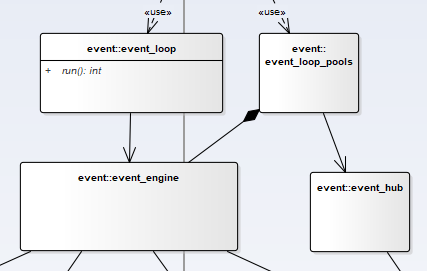


图 事件循环模块类图

事件循环是事件引擎与线程配合，实现事件驱动模型运行的对象。可以看做事件动态的引擎。本模块有两个重要对象，事件循环CEventLoop和CEventLoopPools，二者类似于线程和线程池关系。

* + - * 1. DD-002-001 CEventLoop

CEventLoop事件循环对象本质上是一个线程，线程的主要功能是驱动事件引擎，执行事件循环。class **CEventLoop** : public Pump::Core::**CNonCopyable**, public Pump::Core::Thread::**CThread**;

成员变量

|  |
| --- |
| /\*\*事件循环执行状态\*/  volatile pump\_uint32\_t **m\_dwState**; |
| /\*\*事件循环对象锁\*/  ::Pump::Core::Thread::CMutex **m\_csThis**; |
| /\*\*事件引擎\*/  CEventEngine **m\_objEvEngine**; |

成员方法

|  |
| --- |
| /\*\*启动事件循环  \* 1 启动事件循环业务线程  \* 2 给事件引擎绑定TID  \* 3 向\_\_CEventHub注册事件收集器  \*/  virtual pump\_int32\_t **Start**(); |
| /\*\*终止事件循环  \* 1 终止业务线程  \* 2 从\_\_CEventHub中注销事件收集器  \*/  virtual pump\_int32\_t **Stop**(); |
| /\*\*向事件引擎注册事件\*/  pump\_int32\_t **InsertEvent**(pump\_ev\_handle hEv); |
| /\*\*向事件引擎移除事件\*/  pump\_int32\_t **RemoveEvent**(pump\_ev\_handle hEv); |
| /\*\*向事件引擎注册事件监听器\*/  pump\_int32\_t **InsertListener**(pump\_listener\_handle hListener); |
| /\*\*向事件引擎移除事件监听器\*/  pump\_int32\_t **RemoveListener**(pump\_listener\_handle hListener); |
| /\*\*事件循环线程业务回调，调用引擎的routine()\*/  virtual pump\_void\_t \* **ThreadCallback**(pump\_void\_t \* pData); |

* + - * 1. DD-002-002 CEventLoopPools
        2. DD-002-003 CEventEngine

CEventEngine事件模型驱动引擎，封装三大核心组件，对外提供注册/注销事件、注册/注销监听器接口。

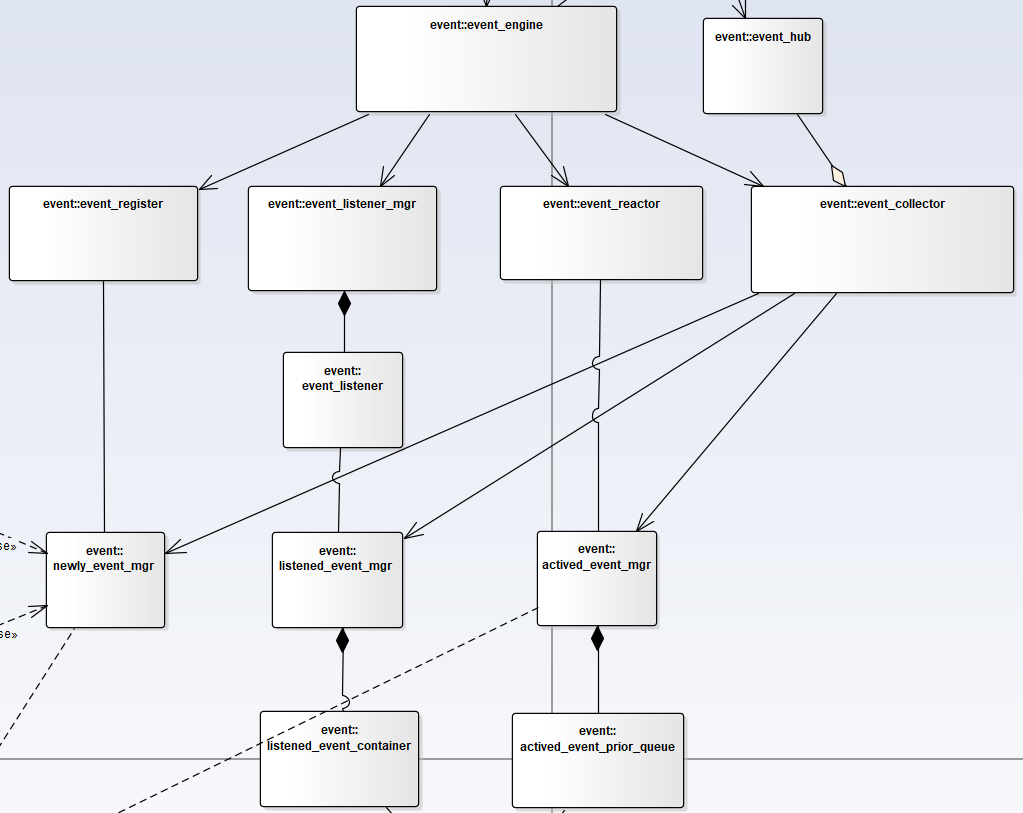
成员变量

|  |
| --- |
| CEventCollector **m\_objEvCollector**; ///< 事件收集器，需要由CEventLoop在启动事件循环时注册到\_\_CEventHub |
| CEventRegister **m\_objEvRegister**;///< 新增事件注册器 |
| CEventListenerMgr **m\_objEvListenerMgr**; ///< 事件监听管理 |
| CEventReactor **m\_objEvReactor;** ///< 事件反应器 |
| pump\_thread\_id **m\_tid**; ///< 绑定事件循环业务线程 |

成员方法

|  |
| --- |
| /\*\*向事件收集器注册事件\*/  int **insert\_event**(CEvent \*pEv); |
| /\*\*向事件收集器移除事件\*/  int **remove\_event(**CEvent \*pEv); |
| /\*\*向监听管理注册事件监听器\*/  int **insert\_listener**(CEventListener \*pListener); |
| /\*\*从监听管理移除事件监听器\*/  int **remove\_listener**(CEventListener \*pListener); |
| /\*\*事件引擎业务函数  \* 1.处理新注册的事件  \* 2.处理处于监听的事件，调用poll\select等  \* 3.处理激活事件  \*/  int **routine**(); |

* + - 1. DD-003 EventEngine 事件引擎模块



* + - * 1. DD-003-001 CEventRegister 事件注册器

事件注册器，存放新加入循环的异步事件，等待引擎按照事件类型进行下一步分发。立即事件放入反应器，条件事件放入对应的监听器。因为需要频繁从新增事件容器存取事件，采用双缓冲技术，避免上锁导致的串行化。

成员变量：

|  |
| --- |
| CEventEngine \*const **m\_pEvEngine**; ///< 绑定事件引擎(非所有者，禁止释放)，构造时绑定 |
| CEventCollector & **m\_refEvCollector**; ///< 绑定事件收集器，构造时绑定 |
| CNewlyEvMgrGuiderForInner **m\_objNewEvMgrGuider**; ///< 新注册事件容器Guider，构造时绑定CNewlyEventMgr |

成员方法

|  |
| --- |
| /\*\*  \* @fn register\_events  \* @return 接口调用状态码  \* @brief 此接口供事件引擎调用,在每轮的事件循环执行周期中,注册Newly事件  \* 1.交换双缓冲中主从缓冲区  \* 2.锁住工作缓冲区  \* 3.遍历Newly事件容器, 根据事件类型加入到对应容器中  \* 3.1 根据事件出发条件, 加入到对应事件管理器  \* 3.2 从Newly事件管理器中移除该事件,更新迭代器  \*/  int **register\_events**(); |

* + - * 1. DD-003-003 CEventListenerMgr 事件监听器管理器

CEventListenerMgr负责统一管理事件监听器。

相关对象：

|  |
| --- |
| /\*\*监听器容器表项\*/  class **CListenerItem**; |
| /\*\*监听器容器类型\*/  typedef std::vector<CListenerItem> **ContainerType**; |
| /\*\*监听器容器迭代器\*/  typedef std::vector<CListenerItem>::iterator **IteratorType**; |
| /\*\*监听器容器查询谓词\*/  class **CFindPredicate**; |

成员：

|  |
| --- |
| CEventEngine \*const **m\_pEvEngine**; ///< 绑定事件引擎(非所有者，禁止释放)，构造时绑定 |
| CEventCollector \***m\_pEvCollector**; ///< 绑定事件收集器(非所有者，禁止释放)，构造时绑定 |
| ::Pump::Core::Thread::CRWLocker **m\_csEvListenerContainer**; ///< 监听器容器的读写锁，因为经常有按照uuid获取监听器的操作，所以采用读写锁提高并发能力 |
| ContainerType **m\_objEvListenerContainer**; ///< 监听器容器 |

方法：

|  |
| --- |
| /\*\*插入新监听器，上写锁\*/  int **insert\_event\_listener**(CEventListener \*pEvListener); |
| /\*\*移除监听器，上写锁\* /  int **remove\_event\_listener(**CEventListener \*pEvListener); |
| /\*\*获取监听器，上读锁，不允许向监听器容器增删\*/  CEventListener \***get\_listener\_by\_uuid\_locked**(const char \*szUuid); |
| /\*\*解锁监听器\*/  void **unlock\_listener\_by\_uuid**(const char \*szUuid); |
| /\*\*执行监听业务\*/  int **listen\_event**(); |

* + - * 1. DD-003-003 CEventListener 事件监听器

监听器负责监听条件事件是否满足激活条件。抽象类。

成员：

|  |
| --- |
| CEventEngine \***m\_pEvEngine**; ///< 绑定事件引擎(非所有者，禁止释放)，构造时绑定 |
| CEventCollector \***m\_pEvCollector**; ///< 绑定事件收集器(非所有者，禁止释放)，构造时绑定 |
| const char \***m\_szUuid**; ///< 监听器uuid用于和CEvent配对 |

方法：

|  |
| --- |
| virtual int **route**(PUMP\_ARG\_IN void \*pArgIn = NULL,  PUMP\_ARG\_IN size\_t dwSizeIn = 0,  PUMP\_ARG\_OUT void \*pArgOut = NULL,  PUMP\_ARG\_OUT size\_t \*dwSizeOut = NULL); |
| virtual int **preListen**(PUMP\_ARG\_IN void \*pArgIn = NULL,  PUMP\_ARG\_IN size\_t dwSizeIn = 0) = 0; |
| virtual int **listen**(PUMP\_ARG\_IN void \*pArgIn = NULL,  PUMP\_ARG\_IN size\_t dwSizeIn = 0) = 0; |
| virtual int **postListen**(PUMP\_ARG\_IN void \*pArgIn = NULL,  PUMP\_ARG\_IN size\_t dwSizeIn = 0) = 0; |

DD-003-003-001 CHandleEventListener 句柄事件监听器

句柄事件监听器监听CHandleEvent及其派生事件。uuid: b8c5ed57-8466-4780-9b88-158bc06191fd。

成员：

|  |
| --- |
| /\*\*  \* 文件句柄IO多路复用后端，<所有者>负责构造和销毁  \*/  ::Pump::Core::Net::CMultiplexBackend \* **m\_pBackend** |
| /\*\*  \* 事件容器，创建事件监听器时，把容器注入到事件收集器CEventCollector  \* <所有者>允许销毁, 创建后需要向 EventCollector 注册  \*/  CMultidimEventContainer \* **m\_pEvContainer** |
| /\*\*  \* 全局句柄事件监听器单例  \*/  static CHandleEventListener \***s\_pEvListener** |

方法：

|  |
| --- |
| /\*\*  \* 获取CHandleEventListener单例指针，调用时创建  \* @param [in] refEvCollector 事件收集器，用于注入监听器的事件容器  \* @param [in] pParam 初始化参数，决定创建IO复用后端  \* @param [in] dwSize 参数数据长度  \*/  static CEventListener \* **getInstance**(  PUMP\_ARG\_IN CEventCollector &refEvCollector,  PUMP\_ARG\_IN void \*pParam = NULL,  PUMP\_ARG\_IN size\_t dwSize = 0  ); |
| /\*\* 直接调用父类的接口 \*/  virtual int **route**(  PUMP\_ARG\_IN void \*pArgIn = NULL,  PUMP\_ARG\_IN size\_t dwSizeIn = 0,  PUMP\_ARG\_OUT void \*pArgOut = NULL,  PUMP\_ARG\_OUT size\_t \*dwSizeOut = NULL); |
| /\*\*初始化监听器，创建事件容器，向事件收集器注册事件容器，创建IO多路复用后端 \*/  virtual int **init**(  PUMP\_ARG\_IN CEventCollector &refEvCollector,  PUMP\_ARG\_IN void \*pParam,  PUMP\_ARG\_IN size\_t dwSize); |
| /\*\*监听前置处理，啥也没干\*/  virtual int **preListen**(PUMP\_ARG\_IN void \*pArgIn = NULL,  PUMP\_ARG\_IN size\_t dwSizeIn = 0); |
| /\*\*监听处理，调用IO多路复用后端，检测文件事件  \* 1.构造::Pump::Core::Net::CFdRetList对象,用于获取发生事件的fd  \* 2.使用多路复用对象获取发生事件的fd  \* 3.for遍历发生事件的fd, 激活其对应的事件对象,并根据事件类型投递到对应的事件管理容器  \* 3.1 在fd监听器的事件容器中查找发生事件的fd,获得FileDescriptor对象  \* 3.2 根据fd事件的状态,激活FileDescriptor关联的事件并处理  \*/  virtual int **listen**(PUMP\_ARG\_IN void \*pArgIn = NULL,  PUMP\_ARG\_IN size\_t dwSizeIn = 0); |
| /\*\*监听后置处理，啥也没干\*/  virtual int **postListen**(PUMP\_ARG\_IN void \*pArgIn = NULL,  PUMP\_ARG\_IN size\_t dwSizeIn = 0); |

* + - * 1. DD-003-004 CEventReactor 事件反应器

事件反应器统一处理处于激活状态的事件。

成员：

|  |
| --- |
| CEventEngine \*const **m\_pEvEngine**; ///< 绑定事件引擎(非所有者，禁止释放)，构造时绑定 |
| CEventCollector & **m\_refEvCollector**; ///< 绑定事件收集器，构造时绑定 |
| CActivedEventMgr &**m\_refActivedEvMgr**; ///< 激活态事件管理器引用，构造时绑定 |

方法：

|  |
| --- |
| /\*\*  \* @return 接口调用状态码  \* @brief 此接口供事件引擎调用,在每轮的事件循环执行周期中,处理Actived事件  \* 1.交换前后台容器  \* 2.锁定双缓冲前台容器  \* 3.for遍历激活优先级队列  \* 3.1 for遍历激活事件容器  \* 3.1.1 调用激活回调函数  \* 3.1.2 清除处理完成的一次性事件  \* 3.2 如果此时激活事件容器仍不为空，则把事件插入到后台容器，等下一轮循环再处理  \*/  int **react\_events**(); |

* + - 1. DD-004 事件管理中心模块

事件管理中心模块定义事件容器相关的接口和对象，包括\_\_CEventHub、EventCollector。

* + - * 1. DD-004-001 \_\_CEventHub

相关对象：

|  |
| --- |
| class **\_\_CHubItem**; |
| #define **ITEMTYPE** pump\_thread\_id, \_\_CHubItem |
| typedef std::pair<**ITEMTYPE**> **ItemType**; |
| typedef std::map<**ITEMTYPE**> **MapType**; |
| typedef MapType::iterator **IteratorType**; |

成员：

|  |
| --- |
| MapType **m\_mapEvCollector**; |
| ::Pump::Core::Thread::CRWLocker **m\_muxMom**; |
| static \_\_CEventHub \* **s\_evHub**; ///< EventHub全局单例 |
| static ::Pump::Core::Thread::CMutex **s\_csEvHub**; ///< EventHub全局构造锁 |

全局方法：

|  |
| --- |
| /\*  \* 向EventHub注册事件收集器，上母写锁->放母写锁  \*/  int **register\_event\_collector**(CEventCollector \*pEvCollector); |
| /\*  \* 从EventHub删除事件收集器，上母写锁->上子锁->放母写锁，然后去除（不允许释放，非事件收集器所有者）  \* @param tid 事件收集器所属CEventEngine绑定的TID  \*/  int **unregister\_event\_collector\_by\_tid**(pump\_thread\_id tid); |
| /\*\*  \* 获取事件搜集器，上母读锁->上子锁  \* @param tid 事件收集器所属CEventEngine绑定的TID  \*/  CEventCollector \***get\_event\_collector\_by\_tid\_locked**(pump\_thread\_id tid); |
| /\*\*  \* 获取事件搜集器，上母读锁->放子锁->放母读锁->放母读锁  \* @param tid 事件收集器所属CEventEngine绑定的TID  \*/  int **unlock\_event\_collector\_by\_tid**(pump\_thread\_id tid); |

* + - * 1. DD-004-002 CEventCollector

事件收集器，一个CEventEngine唯一对应于个CEventCollector。

成员：

|  |
| --- |
| CEventEngine & **m\_refEvEngine**; ///<唯一对应事件引擎，构造时绑定 |
| CNewlyEventMgr **m\_objNewlyEvMgr**; |
| CListenedEventMgr **m\_objListenedEvMgr**; |
| CActivedEventMgr **m\_objActivedEvMgr**; |

方法：

|  |
| --- |
| int **insert\_newly\_event**(CEvent \*pEv); |
| int **insert\_actived\_event**(CEvent \*pEv); |
| int **remove\_actived\_event**(CEvent \*pEv); |
| int **insert\_listened\_event**(CEvent \*pEv); |
| int **remove\_listened\_event**(CEvent \*pEv); |
| int **\_\_process\_sync\_newly\_event**(CEvent \*pEv); |
| int **\_\_process\_async\_newly\_event**(CEvent \*pEv); |

* + - * 1. DD-004-003 CNewlyEventMgr

新增事件管理器，双缓冲技术。

* + - * 1. DD-004-004 CListenedEventMgr

监听事件管理器，存放注册的监听态事件。

相关类型：

|  |
| --- |
| #define **ITEMTYPE** pump\_handle\_t, CListenedEventContainer\* |
| typedef std::map<**ITEMTYPE**> **MapType**; |
| typedef std::map<**ITEMTYPE**>::iterator **IteratorType**; |
| typedef std::pair<**ITEMTYPE**> **ItemType**; |

成员：

|  |
| --- |
| ::Pump::Core::Thread::CMutex **m\_csListenedEv** ///< |
| MapType **m\_mapListenedEv** |

方法：

|  |
| --- |
| bool **insert\_event\_container**(  CEventContainer::EventContainerType emContainerType,  pump\_handle\_t hRealContainer,  const char \*szUuid); |
| bool **remove\_event\_container**(pump\_handle\_t hEvConatiner); |
| bool **insert\_event**(CEvent \*pEv); |
| bool **remove\_event**(CEvent \*pEv); |

* + - * 1. DD-004-005 CActivedEventMgr

激活事件管理器，双缓冲技术。

* + - 1. DD-005 外部接口模块

1. pump\_ac库
   1. 设计思想

使用统一接口实现json、xml、ini、yaml等结构化数据的序列化反序列化功能

上述格式字符串通常用以表达呈树状的结构化数据，因此反序列化之后，数据在内存中的存储结构也是树状的。不用的格式化字符串间的区别就是序列化和反序列化方法不同，一旦解析完毕，在内存中的存储结构时相同的。因此，完全可以利用c++多态特性，抽象出一套统一的接口形式，以实现json、xml、ini、yaml等树形字符串表达格式的列化反序列化功能。

* 1. 详细设计
     1. 类图

计算机生成了可选文字:
] 
丨 丨 
CString 

[图-1]

其中，CDocument表示一份结构字符串文档

* **CDocument**私有成员包含三个重要成员，序列化器CSerializer、反序列化器CDeserializer、树形数据根节点CNode；且序列化器CSerializer、反序列化器CDeserializer都能够操作根节点CNode（不能释放）
* **CNode**是ac框架的核心对象，表示树状结构中的一个子节点，为了适配不同格式化字符串，\_\_CNodeValue用于存放节点的数据源，比如json格式的文档，\_\_CNodeValue的数据源是Json::Value\*；\_\_CNodeRelation用于表达节点间的关系，方便从一个节点快速找到其父子节兄弟点
* **CDeserializer**反序列化器，负责将格式化字符串装换成内存数据结构
* **CSerializer**序列化器，将内存中的数据转化成格式化字符串

CNode组成的树状数据结构在内存中的分布：

计算机生成了可选文字:
listeCNode*» 
， on 鳳 
rant 
parant 
listCCNodeB 
Son list 
parant 
brother 
list€CNodeB 
parant 
brother 

[图-2]

* 1. 反序列化器设计

反序列化器基类

CDeserializer

<- CJsonDeserializer

<- CXmlDeserializer

反序列化流程

计算机生成了可选文字:
构 造 CDocument 指 定 文 档 类 型 
PUMP DOC JSON 
调 用 parse()c str 
调 用 root() 一 > c № d 酽 获 取 根 节 点 

[图-3]

* + 1. json反序列化器、

直接使用jsoncpp作为json反序列化解析器

在CJsonDeserializer::parse()中调用Json::Reader::parse()解析json 字符串

* 1. CNode设计
     1. 节点查找

因为使用了三方的解析模块，因此解析完成后，内存中并没有如[图-2]形成CNode组成的树状数据结构

为了实现统一接口的目标，需要将三方解析模块中的数据结构转换成CNode组成的树状数据结构

采用的方案是，采用按需加载的策略，当用户读取某一个节点的数据，就提前完成它下一层子节点的加载，如下图所示：

计算机生成了可选文字:
GetF i r stSonNode （ ） 
Son ist 
0u0 」 e 
parant 
listeCNode*» 
50n IISt 
list(CNode*:. 
parant 
brother 
0 ， e 
brother 
list(CNode*:. 
0 ， e 
parant 
parant 
brother 

[图-4]

当用户调用CNode::GetFirstSonNode(pNode)获取某个节点的一个子节点时，如果该节点是Object类型，CNode会再次遍历其对应Json::Value的子节点，然后逐个创建CNode并加入到树形结构中。延迟加载的方案，使得不需要一次性将三方解析模块中的数据结构转换成AC的树形结构数据。如图所示浅绿色的两个子节点就是读取其父节点时提前加载的。

* + 1. 节点增加

用户调用AC模块的接口，向CNode树增加节点，很容易实现。其中需要注意的，任然是使用三方解析模块的情况。此时添加CNode节点的同时，还需要向三方解析模块同步增加数据，以免数据不一致。

* + 1. 节点删除

删除某一个节点，当前方案会连同删除其所有子节点，否则容易造成内存泄露。

同时别忘记在使用三方解析模块时，同步删除模块中的数据

1. 插件化程序框架
2. 附件

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |